



ENHANCING THE NS-2 NETWORK SIMULATOR  
FOR  
NEAR REAL-TIME CONTROL FEEDBACK  
AND  
DISTRIBUTED SIMULATION  
THESIS

John S. Weir, Major, USAF  
AFIT/GE/ENG/09-47

DEPARTMENT OF THE AIR FORCE  
AIR UNIVERSITY

**AIR FORCE INSTITUTE OF TECHNOLOGY**

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GE/ENG/09-47

ENHANCING THE NS-2 NETWORK SIMULATOR  
FOR  
NEAR REAL-TIME CONTROL FEEDBACK  
AND  
DISTRIBUTED SIMULATION

THESIS

Presented to the Faculty  
Department of Electrical and Computer Engineering  
Graduate School of Engineering and Management  
Air Force Institute of Technology  
Air University  
Air Education and Training Command  
In Partial Fulfillment of the Requirements for the  
Degree of Master of Science in Electrical Engineering

John S. Weir, BSCE, MSEM  
Major, USAF

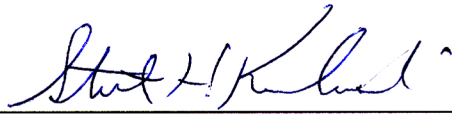
March 2009

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

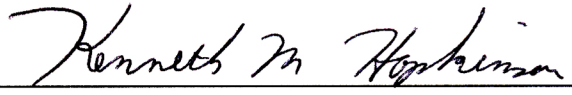
ENHANCING THE NS-2 NETWORK SIMULATOR  
FOR  
NEAR REAL-TIME CONTROL FEEDBACK  
AND  
DISTRIBUTED SIMULATION

John S. Weir, BSCE, MSEM  
Major, USAF

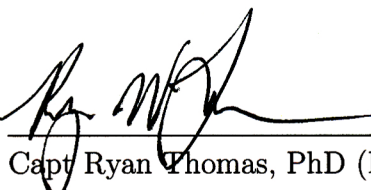
Approved:

  
Lt Col Stuart Kurkowski, PhD (Chairman)

26 Feb 09  
Date

  
Dr. Kenneth M. Hopkinson (Member)

February 26, 2009  
Date

  
Capt Ryan Thomas, PhD (Member)

26 Feb 09  
Date



## *Abstract*

A network simulator coupled with a visualization package enables the human visual system to analyze the results of network modeling as a supplement to analytical data analysis. This research takes the next step in network simulator and visualization suite interaction. A mediator (or run-time infrastructure (RTI) in the literature) provides researchers the potential to interact with a simulation as it executes. Utilizing TCP/IP sockets, the mediator has the capability to connect multiple visualization packages to a single simulation. This new tool allows researchers to change simulation parameters on the fly without restarting the network simulation.

## *Acknowledgements*

I would like to thank my thesis advisor Lt Col Kurkowski for the vision of this research as well as the motivation he provided throughout it's course. This work could not have been as successful without the work of Josh Abenathy on the NetVis project. I would also like to thank my wife and two children for their understanding and patience which allowed me to concentrate on this work.

John S. Weir

# *Table of Contents*

	Page
Abstract . . . . .	iv
Acknowledgements . . . . .	v
List of Figures . . . . .	x
List of Abbreviations . . . . .	xii
I. Introduction . . . . .	1
II. Network Simulation and Visualization with NS-2 . . . . .	5
2.1 Network Simulators . . . . .	5
2.2 Simulation Usage . . . . .	7
2.2.1 Traditional Simulation Usage . . . . .	7
2.2.2 Emulation . . . . .	8
2.2.3 Co-Simulation . . . . .	9
2.3 Network Visualization . . . . .	10
2.4 Other Efforts . . . . .	11
2.4.1 User Interaction and Command Feedback Design . . . . .	11
2.5 Summary . . . . .	11
III. Command Feedback Design and Implementation . . . . .	13
3.1 Software Design . . . . .	14
3.1.1 Communication Mediator . . . . .	18
3.1.2 Simulated Visualization . . . . .	18
3.1.3 Command Reception and Handling . . . . .	19
3.1.4 Modified Simulation Execution . . . . .	20
3.2 Software Implementation . . . . .	21
3.2.1 Communication Mediator . . . . .	21
3.2.2 Simulated Visualization . . . . .	24
3.3 Additional NS-2 Enhancements . . . . .	28
3.3.1 Clock Time Scheduler . . . . .	28
3.3.2 Synchronized Scheduler . . . . .	30
3.4 NetVis Enhancements . . . . .	32
3.4.1 Command Feedback . . . . .	33
3.4.2 Clock Synchronization . . . . .	34
3.4.3 Co-Simulation with SimVis . . . . .	34

	Page
IV. Improving Performance and Usability . . . . .	36
4.1 Additional Enhancements . . . . .	36
4.1.1 TCL Script Enhancement . . . . .	36
4.1.2 NS-2 Enhancement . . . . .	37
4.1.3 Tool Enhancement . . . . .	37
4.1.4 Configurable Command Tool . . . . .	38
4.2 Testing and Evaluation . . . . .	40
4.2.1 Mediator Testing . . . . .	41
4.2.2 Testing the NS-2 Data Connection . . . . .	42
4.2.3 Testing Simulation Data with SimVis . . . . .	42
4.2.4 Command Testing . . . . .	43
4.2.5 Testing with NetVis . . . . .	45
4.2.6 Scenario Testing . . . . .	46
4.2.7 Distributed Hybrid system Testing . . . . .	52
4.2.8 Educational System Testing . . . . .	53
V. Contributions and Future Work . . . . .	56
5.1 Command Feedback Network Simulation System Components . . . . .	56
5.1.1 Mediator . . . . .	57
5.1.2 NS-2 . . . . .	57
5.1.3 Configurable Command Tool . . . . .	58
5.1.4 NetVis . . . . .	58
5.2 Contributions . . . . .	58
5.2.1 Command feedback . . . . .	58
5.2.2 Distributed Hybrid System Capability . . . . .	60
5.2.3 Synchronized System Execution . . . . .	61
5.3 Future Work . . . . .	64
5.3.1 Refactoring . . . . .	64
5.3.2 Lazy Initialization . . . . .	64
5.3.3 Command String Parsing . . . . .	64
5.3.4 Command Transmission . . . . .	65
5.3.5 NetVis Synchronization . . . . .	65
5.3.6 Additional Development . . . . .	65
Appendix A. System Requirements and Startup Procedures . . . . .	67
A.1 System Requirements . . . . .	67
A.1.1 Hardware . . . . .	67
A.1.2 Operating Systems . . . . .	67
A.1.3 Java Virtual Machine . . . . .	68
A.1.4 Integrated development environment . . . . .	68

	Page
A.2 IP configuration . . . . .	68
A.3 System Startup and Operation . . . . .	69
Appendix B. NS-2.32 Code Changes and Building Instructions . . . . .	70
B.1 Changes to Makefile.in . . . . .	70
B.2 Changes to Makefile . . . . .	70
B.3 Addition of Command Controller Package . . . . .	70
B.4 Changes to the Common Folder . . . . .	71
B.5 Changes to the tcl\lib Folder . . . . .	71
B.6 Boost Library Threading . . . . .	71
Appendix C. Interface Document . . . . .	72
C.1 Sockets . . . . .	72
C.2 Protocol . . . . .	72
C.2.1 Connection Protocol . . . . .	72
C.2.2 Disconnection Protocol . . . . .	73
C.3 Synchronization . . . . .	73
C.3.1 ClockTimeScheduler . . . . .	73
C.3.2 VisSyncScheduler . . . . .	73
C.4 Commands . . . . .	74
Appendix D. Commands Developed . . . . .	75
D.1 Wired Commands . . . . .	75
D.1.1 change_queue_size . . . . .	75
D.1.2 turn_off_cbr . . . . .	75
D.1.3 turn_on_cbr . . . . .	75
D.1.4 change_cbr_packetSize . . . . .	76
D.1.5 change_cbr_interval . . . . .	76
D.1.6 move_cbr . . . . .	76
D.2 Wireless Commands . . . . .	77
D.2.1 move_wireless_node . . . . .	77
D.3 Non-specific Commands . . . . .	77
D.3.1 pause . . . . .	77
D.3.2 resume . . . . .	77
D.3.3 update_vis_clock . . . . .	78
D.3.4 set_read_ahead_offset . . . . .	78

	Page
Appendix E. Command Development . . . . .	79
E.1 Developing commands . . . . .	79
E.1.1 CommandParser code . . . . .	80
E.1.2 CommandHandler code . . . . .	81
E.2 Naming conventions . . . . .	82
E.3 Testing commands . . . . .	83
E.3.1 TCL script testing . . . . .	83
E.3.2 CCT testing . . . . .	83
E.3.3 NetVis testing . . . . .	84
Bibliography . . . . .	85
Index . . . . .	87
Author Index . . . . .	1

## *List of Figures*

Figure		Page
1.1	Illustration of an educational scenario for simulation. . . . .	3
2.1	Example NS-2 chronological TCL script. . . . .	7
3.1	Mediator depiction with NS-2 and visualizations. . . . .	15
3.2	Commandhandler class design with Listener and CommandParser classes. . . . .	16
3.3	CommandHandler Sequence diagram of a StopCBRCommand. . .	17
3.4	Illustration of command flow from visualization software through the mediator to NS-2. . . . .	19
3.5	Illustration depicting the two required TCP/IP connections required between the mediator and NS-2. . . . .	22
3.6	Snapshot of the Mediator GUI . . . . .	23
3.7	Depiction of the Simulated Visualization GUI . . . . .	25
3.8	Example TCL Script for adding a TCP/IP connection. . . . .	26
3.9	Example TCL script for testing CommandHandler code. . . . .	26
3.10	Scheduler design concept illustrating inheritance of the ClockTime and VisSync schedulers. . . . .	29
3.11	Depiction of the ClockTime scheduler dispatch logic. . . . .	30
3.12	Depiction of the VisSync scheduler dispatch logic. . . . .	31
3.13	Image of NetVis depicting the newly designed command panels . .	33
4.1	Example TCL script for TCP/IP connection with the mediator. . .	36
4.2	Example configuration file for the Configurable Command Tool. . .	37
4.3	Depiction of the Command Configuration Tool GUI. . . . .	39
4.4	Illustration of the Configurable Command Tool as an instruction tool.	40
4.5	Example TCL Script for CommandHandler Development. . . . .	44
4.6	Example TCL Script for invoking a ClockTime or VisSync scheduler.	45
4.7	Illustration of a wired scenario utilized for testing. . . . .	47
4.8	Illustration of a wireless scenario utilized during testing. . . . .	49

Figure		Page
4.9	Illustration of a CAOC scenario utilized during testing. . . . .	51
4.10	Illustration of the Hybrid test system. . . . .	52
4.11	Illustration of an educational system during testing. . . . .	54
5.1	Depiction of the research system components. . . . .	56
5.2	Photo of the Hybrid test system. . . . .	60
5.3	Photo of an educational simulation playback. . . . .	62
E.1	Depiction of command inheritance. . . . .	79
E.2	Example code for command development. . . . .	80
E.3	Example command code requiring multiple setArgs() and scheduled handlers. . . . .	81
E.4	Example handler code used to create dynamic TCL. . . . .	82
E.5	Example command.cfg code for the Configurable Command Tool. .	83



## *List of Abbreviations*

Abbreviation		Page
NS-2	Network Simulator-2 . . . . .	5
OPNET	Optimized Network Evaluation Tools . . . . .	5
GTNetS	Georgia Tech Network Simulator . . . . .	5
GloMoSim	GlobalMobile Information Systems Simulation Library . . . . .	5
TCL	Tool Command Language . . . . .	6
NAM	Network Animator . . . . .	6
cbr	Constant Bit Rate . . . . .	8
iNSpect	Integrated NS-2 Protocol and Environment Confirmation Tool	10
TCP/IP	Transmission Control Protocol/Internet Protocol . . . . .	14
SimVis	Simulated Visualization . . . . .	18
CCT	Configurable Command Tool . . . . .	38

# ENHANCING THE NS-2 NETWORK SIMULATOR FOR NEAR REAL-TIME CONTROL FEEDBACK AND DISTRIBUTED SIMULATION

## I. Introduction

Simulation is an alternative to the expense of prototyping equipment. It is also a non-evasive method to test new software without risking an operational system. Simulation allows a user to change the behavior of an item by replacing algorithms or other software aspects. Output data allows a quick review of the change's outcome. Simulation is used extensively in many applications with software intensive components. Networking is one of the areas where extensive research and development is explored, validated, and verified by simulation.

Network simulation provides researchers with a tool to analyze and evaluate proposed design solutions for many aspects of networking. Simulation allows users to test and evaluate new ideas without producing new hardware, deploying new software, or impacting existing networks. Simulation can also allow the integration of prototype hardware to determine performance parameters through emulation. Simulation is a cost-effective tool that is constantly getting more powerful and realistic as research progresses and improvements are made to propagation models, and other aspects of network simulation.

Network simulation has impacted the development of many aspects of networking. Research is ongoing to improve many aspects of network simulation such as new protocols, new algorithms, and new parameter thresholds. New environments are developed, models are improved to be more realistic, and simulations are coupled together to represent larger systems, adding to the complexity of network simulation.

Most simulators have an integrated visualization capability or else one is available in an additional software suite. This visualization capability provides a tool for analysis and allows visual demonstrations of results. Wired and wireless networks can be rendered and all aspects of the simulations from packets to satellites can be visually represented with images to improve network representation. Information visualization and the representation of the simulated network are targets of constant improvement. Even with these improvements, the focus has been on presenting the information.

A majority of the current visualization suites coupled with a network simulator, are deployed after the simulation execution has completed. Most network simulation visualizers post-process information and so they present limited opportunities for interactive user controls. As such, researchers must run multiple iterations of a scenario to make changes to the execution and eventually conclude that the proposed modifications are successful or unsuccessful. This restrains the usefulness of the simulation and visualization and increases research time. What if the user could interact with the network simulation as it executes? What if the interaction was controlled by direct manipulation of the visualization suite? A vast improvement in research capability could be achieved. Users could see the effects of changes made to the network simulation in near wall-clock-time and have the ability to change additional parameters or remove their changes with the click of a mouse. The ability to interact would allow multiple iterations of test parameters in one simulation run. The possibility to narrow searches for correct algorithms, validate specific weaknesses or concerns would become easier to accomplish. Time savings would be a benefit of this concept, due to the user input instead of multiple “scripted” iterations.

What if the simulators could be linked to multiple visualizations? Not only is there potential benefit to research, but also in training and education. Simulation can be used to demonstrate key concepts or methods by highlighting the events on the visualization. Training by simulation would provide trainees the possibility to execute “what if” scenarios without detrimental effects on an operational system. User

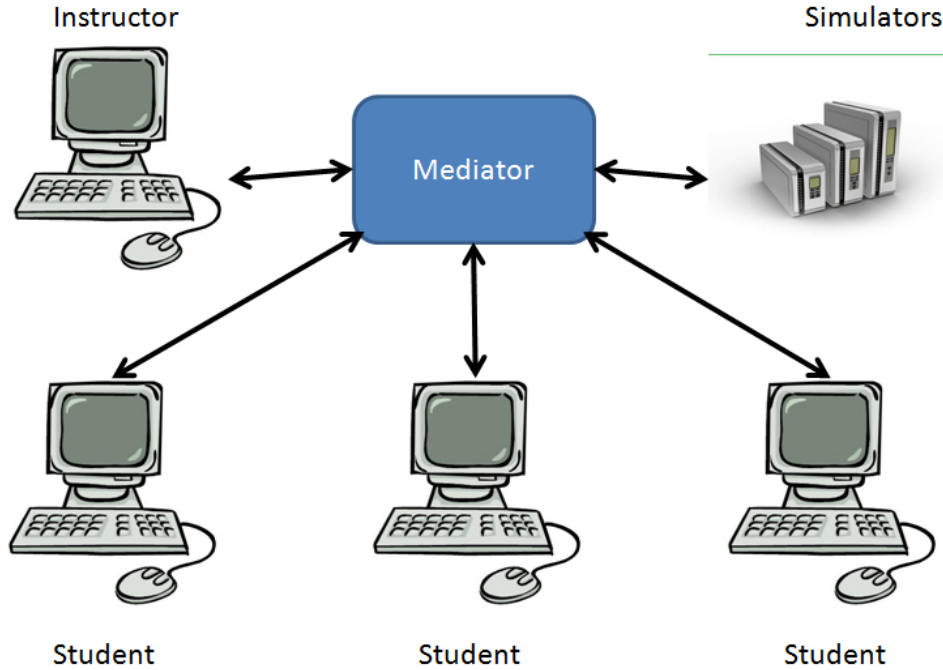


Figure 1.1: Illustration of an educational scenario for simulation. A mediator brokers data from the simulators to the instructor and students as well as user interaction in reverse order from student and instructor to the simulations.

interaction with a running simulation would add another dimension to education and training. This additional capability would allow instructors and/or students coupled with a specific scenario to achieve a training goal through interaction. Either a previously recorded simulation can be read from a file as a static simulation or an active simulation can be utilized with the mediator facilitating instructor and/or student interaction. Figure 1.1 illustrates such a system with multiple students watching or interacting with a simulation as it progresses.

The authors of [27] proposed real-time simulation running concurrently with actual network traffic to allow network managers to provide “what-if” scenarios to improve customer satisfaction and network utilization rates. However, their research does not change simulation parameters on the fly and requires the same paradigm of running the simulation multiple times with different parameter settings to achieve the best customer satisfaction rates when human interaction is used.

This research proposes a change in the use of simulation visualization by providing user interaction. This will provide near clock-time response to user inputs on the screen of a network visualization and allow multiple visualizations linked directly to a network simulator. This capability will greatly enhance research capability in network simulation as well as reduce the time required to perform the research.

## II. Network Simulation and Visualization with NS-2

Network simulation is utilized in research and training environments. Simulation is a useful method to test and evaluate new network protocols and equipment. Many tools, proprietary and open source are available for these uses. Components required to successfully simulate a network include tools to build the network layout, simulate the network's performance, visualize the network simulation, and analyze the results. We are not concerned with network or scenario creation, but focus on the simulator and visualization components required to successfully simulate a network scenario.

### 2.1 *Network Simulators*

The key element to perform any simulation is the simulation software. A sampling of current network simulation software includes NS-2 [25], OPNET [20], GT-NetS [22], OMNeT++ [1], and GloMoSim [24]. Each instance of network simulation software allows the creation of network topologies either utilizing visualization software, a script file, or a combination of the two. The true role of the network simulation software is the execution of the network events using the previously created topology. These network events adhere to a set of rules generated with protocols and agent properties. Network simulators are usually standalone entities, however, coupling with visualization software can improve the understanding of network events and makes the simulation a more powerful tool for research.

Each of the simulators listed above provides propagation models, protocols, and wireless or wired network behaviors to support many topologies. These simulators allows researchers to include anything from workstations to satellites. Output data can be tailored to suit the research performed. Traditional use of simulation software is implemented by running a set of design parameters entered in the simulation script, the network simulator then steps through processing the events, afterwards the output data is reviewed and analyzed to determine the impact of the system under test.

For example, if a certain protocol achieved a high delivery ratio under the tested conditions.

This research will focus on Network Simulator 2 [25]. NS-2 is an open-source network simulator package used extensively in research programs. NS-2 utilizes a scripting language Tool Command Language (TCL) to execute simulations. The scripting language allows the user to quickly generate new simulation scenarios without the burden of compiling the NS-2 C++ code. This makes the simulation suite a very flexible tool, however, it complicates the software structure by creating two layers of code that are linked together by dual language objects existing in both the TCL and C++ code.

NS-2 is a research project in itself, and its extensive use during the past decade [8] has produced many extensions and tailored applications that are publicly available. These enhancements can be utilized as templates to develop new applications for network simulation. NS-2 also has a follow-on development currently under development, NS-3 [8]. NS-3 will continue to utilize script-based simulation while providing better wireless models and improved scalability. [8]. It is unknown at this time if this research will be portable to the NS-3 platform.

NS-2 is a discrete event simulator, and thus executes events in a sequential manner. The events are “scheduled” for execution in a chronological manner based on the type of scheduler specified in the TCL script. As the events execute, a trace file can be used to log the data describing the specific events. When a Network Animator (NAM) trace file is specified in the TCL script, a specific formatting is used to allow NAM to playback the events that occurred during the simulation. Discrete event simulation is useful, but requires multiple iterations of simulation runs to accomplish “what if” scenarios. No previous research was discovered that implemented dynamic execution allowing network changes to occur while the simulation runs.

NS-2 is a powerful tool for network simulation. It is extremely flexible due to the accessibility of its source code and simplicity of code enhancement or modifications.

A wealth of previous enhancements are also readily available on the Internet as well as documentation and specific instructions written by previous developers [16].

## 2.2 *Simulation Usage*

The power of network simulation is its flexibility, it can be extended to include new protocols, agents, or additional classes. The potential exists to extend and tailor classes to test new methods of networking. Not only can the software be enhanced, hardware can also be introduced into the simulation to interact with software events. This allows prototype hardware testing as well as the ability to evaluate realistic network applications. Additionally, other simulations can be run simultaneously where networking is only a portion of the total simulation environment running in a co-simulation environment. These three uses of network simulation are described here.

```
set ns [New Simulator]
...
#configure nodes
...
#schedule events
$ns at 0.5 "$cbr0 start"
$ns rtmodel-at 1.0 down $n(1) $n(2)
$ns rtmodel-at 2.0 up $n(1) $n(2)
$ns at 2.5 "$cbr0 stop"
```

Figure 2.1: Example NS-2 TCL script detailing chronologically scheduled events predetermined before execution. The “at” in each line is the a priori established event time.

*2.2.1 Traditional Simulation Usage.* Traditionally, internal design modifications (i.e. new protocols or agents) are written in the simulator’s source code (i.e., C++ for NS-2). Then, a simulation is performed by describing a scenario, to include node characteristics, traffic characteristics (e.g., size, timing, and distribution), and protocol characteristics. For example, when using NS-2, a TCL script is written to generate the scenario. The TCL script contains code that creates nodes, links between the nodes, and agents that produce a specific type of traffic (i.e. UDP, TCP/IP, and



FTP) as well as defining parameters specific to each type of network item. After the network items are defined, the script is developed to create specific events by scheduling these events to occur at the specified time. The seventh line of Figure 2.1 takes the link between nodes one and two down at time 1.0 seconds. All node movement and other behaviors are defined via the TCL script.

This scenario is then provided to the simulator for scheduling with specific timing events, such as a link going down, etc. The NS-2 simulation executes the TCL script, and as a discrete event simulator, NS-2 schedules the events in chronological order. Figure 2.1 depicts a traditional NS-2 TCL script, where constant bit rate (cbr) traffic is started and stopped and a link between nodes one and two is brought down and then back up. These events are preconceived and are scheduled prior to the start of the simulation. To change a scheduled (e.g., a “what if”) input one must wait until the simulation completes or is stopped by user intervention. Then the TCL script is changed and re-run to produce the new data. If the source of the behavior of note is not contained within the TCL script, the C++ code must be modified and recompiled, then the TCL script can be re-run.

*2.2.2 Emulation.* The NS-2 simulator provides the opportunity to emulate all or a portion of the simulation. Emulation provides the ability to introduce hardware into the simulation, producing the capability to inject live traffic into NS-2 while the simulation is executing. It also generates live traffic as output from the simulation back to the hardware-in-the-loop. There are two modes available for emulation in NS-2, Opaque mode and Protocol mode. Opaque mode where actual packets are not modified by the simulator. In Protocol mode, on the other hand, the actual packets generated by the hardware are interpreted and potentially modified by the simulation. Emulation within NS-2 is sometimes considered real-time, however, events must be executed sequentially in a computing environment. With the limitations of serial execution, real-time execution is unrealistic.

Emulation also uses a real-time scheduler separate from the main simulation thread/s non-real-time scheduler. The real-time scheduler tries to execute network events at *actual* moments in time [12]. The emulation environment interfaces with hardware through tap agents and network objects.

Emulation is a target for improvement within the research community. The emulation module in NS-2 has weaknesses, especially with wireless emulation, and can lead to unpredictable timing behaviors. In [12], the system call for the time is replaced by a function that uses the CPU cycle counter to measure times available on most CPUs. The only restriction to this use is that the CPU has to run at a constant frequency so time measurement stays constant. [12] attempts to eliminate the most frequently used system call by introducing a precise method for executing waiting events.

*2.2.3 Co-Simulation.* Co-Simulation couples multiple simulators in one run-time environment. This is a very useful approach to solve larger problems that a single simulation cannot address. However, difficulties arise with simulation communication and clock synchronization [6]. The introduction of a mediator to synchronize events is often used. Branicky et al [5] outlines a framework for co-simulation with a networked control system providing control and feedback scheduling. [9] allows users to investigate electromagnetic scenarios involving communication by linking electromechanical transient simulators with NS-2 using a simulation synchronization and communication router *RTI*. Our research springboards off the co-simulation concept by offering the potential for co-simulation as well as simultaneous visualization of the aggregate simulations.

Co-Simulation usually requires some method of synchronization to ensure the components work efficiently together. EPOCHS [9] uses a run-time interface (RTI) to “glue” the components together. This component is responsible for all communications routing and simulation synchronization. In [5], a controller is described to control the connection of multiple “plants” that would operate in a co-simulation en-

vironment. This research will investigate co-simulation mediation of communication and synchronized execution of separate software components.

### ***2.3 Network Visualization***

Most network simulators have a visualization capability either integrated or in the form of an additional software suite. Visualization adds another dimension to network analysis by complimenting the statistical data produced by the simulation. The true power of visualization, however, is the ability to ‘playback’ the network events generated by the simulation. Visualization software can render packets, queues, nodes, and other network entities and events.

The visualization of a network simulation allows the user to analyze the chain of events recorded by the previously run simulation software. This helps pinpoint problems and narrow the amount of statistical data the researcher must analyze to determine the effects of the scenario. Visualizations also allow the user to quickly “fast forward” through the simulation time line to reach a point of interest as well as “rewind” to view behaviors again and again.

For NS-2 there are at least three visualization packages; the Network Animator (NAM) [26], the integrated NS-2 protocol and environment confirmation tool (iNSpect) [10], and the Network Visualizer (NetViz) [3]. All three visualization suites have their own benefits, but operate very similarly. The visualization software parses a trace file constructed during NS-2 execution, translates the data, and renders the information to a monitor. NAM has the ability to use streams for real-time applications utilizing `stdin` and additional trace file data [7], but does not provide the capability for user interaction with the simulation. Currently, neither iNSpect nor NetViz can process anything other than a trace file after the simulation is complete. OPNET’s visualization software provides a post simulation analysis as well.

## 2.4 *Other Efforts*

Discrete event simulation has its place in research, but this beneficial concept can be improved upon. By adding flexibility into a simulation, where not all events are scripted, great potential exists to enhance network research. If a user is given this ability, the benefit could reach many different situations.

*2.4.1 User Interaction and Command Feedback Design.* A beneficial addition to the functionality of a network simulator is the ability to control the simulation as it executes. There is great potential for decreased research time as well as the possibility to execute multiple “what if” scenarios during the course of a single simulation instance. Instead of hundreds of consecutive or parallel simulations, a researcher can narrow down possibilities within a handful of executions.

Not only will this benefit research, but also education and training. Students can now interact with a pre-scripted scenario to test their abilities to control a situation. Instructors can also interact with the simulation to induce problems (i.e. take down a link) for the students to react to. Many applications could benefit from the ability to interact with a running simulation.

## 2.5 *Summary*

A network simulation system can be comprised of the simulator and an optional visualization software suite. The simulator has the capability to perform exclusively or with other simulators in a co-simulation. Hardware can also be introduced into the system through the use of emulation. The visualization provides a means to “see” the network events and objects. A researcher has many tools and software suites at their disposal to help accomplish the end goal.

With traditional simulation, a researcher executes a simulation of a priority established list of events and analyses the data possibly with the help of visualization software. The visualization is employed *after* the simulation has completed. This

research will take the next step by defining interactive simulation and dynamic event execution allowing visualization of the simulation as it executes.

### III. Command Feedback Design and Implementation

User-simulator interaction is the key driver of this research. It provides the capability for a user to execute a scenario consisting of a network simulator with a visualization tool. The visualizer has the ability to feed commands back to the simulation *while it is executing*. This brings a previously static simulation to a new level by introducing the ability to change parameters on the fly without multiple iterations of execution. Researchers now have the power to perform “what-if” scenarios without modifying the simulation script and rerunning the model. This new flexibility in simulation has the potential to save countless hours of waiting on the simulator to complete before performing analysis.

Most network simulators are sequential and require multiple iterations of the code to solve a problem in all its complexity. An improved method would allow user changes to the simulation as it runs. A traditional simulation schedules events in a chronological order and produces a non-dynamic execution following the pre-conceived events, randomness withstanding. Our design takes a different approach to the problem. By introducing user interaction, events are created as the simulation executes, providing dynamic simulations. This allows the usage of a very basic script while the user provides tool commands through an interactive visualization.

The design and implementation of command feedback for NS-2 application can be divided to ease the software development process. The three major development efforts for our research are: data and command communication mediator, simulated visualization, and command reception and handling. Figure 3.1 illustrates the three components of development. The mediator provides communication brokering between the simulator and the visualization. The simulated visualization provides the capability to receive simulation data and transmit commands to the simulator through the mediator. The command and reception and handling are contained within NS-2 code and provide additional functionality that receives and executes commands from the visualization as well as transmitting data to the visualization through the medi-

ator. Software design of the major components was an essential step in this research development.

### ***3.1 Software Design***

Software engineering practices were followed utilizing software patterns, refactoring, and abstract classes when applicable. The result is a modular design that simplifies further efforts to produce additional functionality and allows refactoring or rewriting of modules without disrupting the functionality of other modules.

Interaction between a network simulation and visualization tool could be accomplished by modifying the software in each element to communicate with each other directly. This requires application specific re-coding without the potential to integrate additional simulators or multiplex communication. This research takes a different approach by introducing a mediator to broker the communication and minimize the simulation/visualization code changes. This concept allows the integration of many simulators and visualizations as well as the capability to multiplex simulation data to more than one connected visualization. Simulation and visualization software can be modified to work with the universal mediator. The mediator provides Transmission Control Protocol/Internet Protocol (TCP/IP) connection service for simulators and visualizers. This design brings the added benefit of allowing multiple visualizers to connect to a single simulator. In this case, the network simulator acts as the server and the visualizers act as clients. Figure 3.1 depicts the flow of events between the server and client in a typical application. The mediator is the heart of the application, collecting commands from each connected visualization and passing them to the simulator. The mediator also receives the data from the simulation and duplicates it for each visualizer, allowing multiple clients to see the same simulation concurrently. The visualization passes the command through the mediator to the simulation, which handles the commands as received. The simulator translates the commands, schedules them at the current simulation time, then when the event is handled, it is passed

to the TCL instance for dynamic code execution. The mediator also provides the potential to link multiple simulations and visualizations increasing it's usefulness.

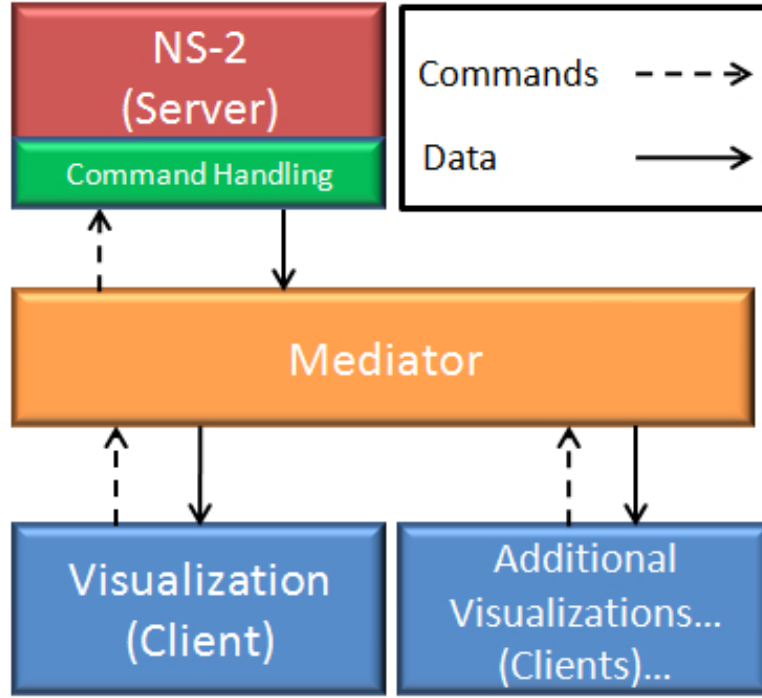


Figure 3.1: Mediator depiction showing data and command flow between the NS-2 instance and visualizations. Data is duplicated when multiple visualizations are connected.

Design of the software stems from a mixture of the research interest and the existing design of NS-2 itself. The basic sequence of communication would be data flow from the NS-2 execution to the visualization for rendering on a monitor and command flow from the visualization to NS-2 for execution parameter changes. NS-2 executes through a TCL script which utilizes the C++ classes of NS-2 itself. The NS-2 classes that exist in both worlds are known as dual objects. One dual object was developed for this research, the Listener class. The Listener class was designed to envelope all command reception and handling within NS-2. It executes in parallel with the main NS-2 functions, requiring it's own thread. This thread monitors a TCP/IP socket for commands passed from the visualization through the mediator. The CommandParser is contained within the Listener. As depicted in Figure 3.2 the CommandParser holds



singleton instances of each defined CommandHandler. The CommandParser is also the only object that has access to the NS-2 scheduler instance.

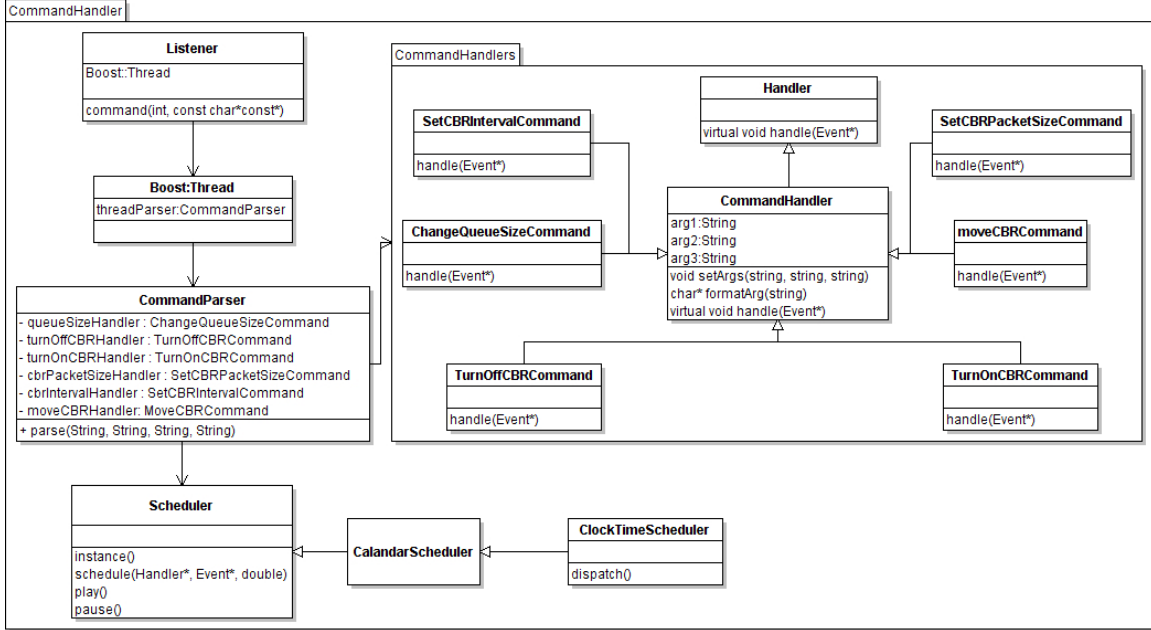


Figure 3.2: CommandHandler class design depicting the Listener and Command-Parser classes. The CommandHandler class is abstract and six subclasses are shown each with their own handle() procedure.

Figure 3.3 illustrates the typical execution flow for command reception, specifically for a Stop CBR Command. Object creation begins with the TCL script. As the TCL script is parsed, a SocketListener object is created within C++. The SocketListener object creates a new Boost::Thread [4] and passes the routine for the thread to execute (threadInstructions). The thread creates a new CommandParser object which builds a singleton instance of each command handler (StopCBRCommand depicted). After all objects are instantiated, the thread opens a TCP/IP socket and connects to the mediator using the protocol “command”. The thread then listens to the socket for any commands sent from the mediator (passed on from the visualization or SimVis). After a string is received, the thread parses it into four separate strings. Each command string consists of the command itself and three strings for arguments. In Figure 3.3 a StopCBRCommand is received. The command and three arguments are passed to the parse method of the CommandParser object. The CommandParser is the ob-

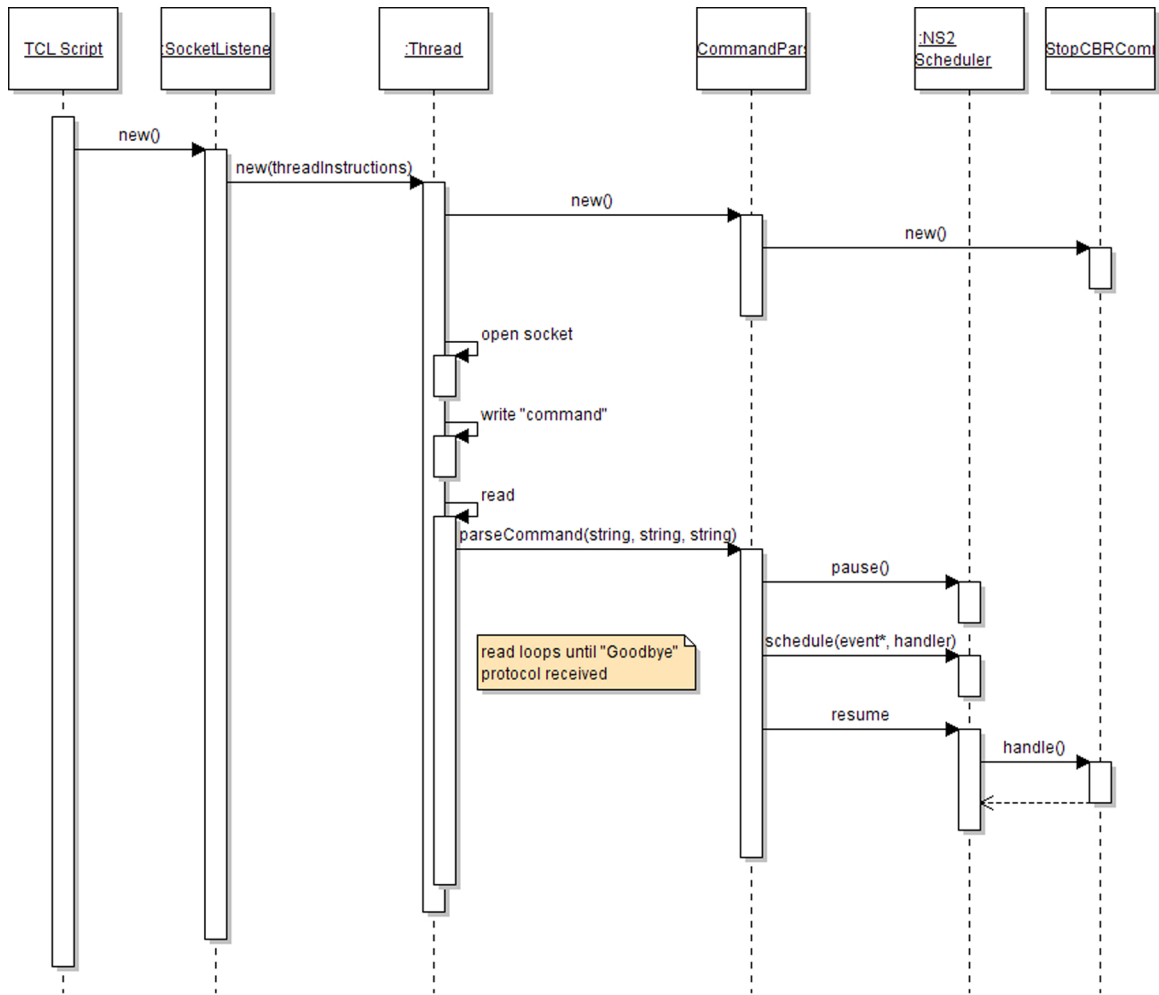


Figure 3.3: CommandHandler Sequence Diagram depicting object initiation from the TCL Script to C++ and execution of a StopCBRCommand command within NS-2. The command is initiated from a visualization with command feedback capability.

ject that directly interacts with the NS-2 scheduler. The scheduler is first paused to stop the simulation, allowing the scheduling of a new event. The CommandParser then schedules the new event at the current simulation time using the command object as the handler (StopCBRCommand). After the event is scheduled, the scheduler is resumed allowing NS-2 execution to continue. When the scheduler calls the event, it calls the command's handler code which is specific to each of the commands developed. This execution sequence allows the CommandParser to schedule the command

handling at or near the top of the scheduler's stack ensuring immediate command response.

*3.1.1 Communication Mediator.* To ease the design of the software to perform these actions, a communication facilitator was developed to broker the communication flows from NS-2 to the visualization and back. This broker is the communication mediator and will be referred to as the *mediator* in the rest of this text. The mediator performs the function of connecting simulators (servers) and visualizations (clients). A TCP/IP connection is used for each of the servers or clients connected to the mediator. Since TCP/IP connections are full duplex, this allows commands and data to pass over the same connection. A simple protocol will be used for connection and disconnection from the mediator. The initial design allows multiple connections of servers and clients to allow additional capability to researchers. The mediator concept is depicted in Figure 3.1.

TCP/IP and Java were a design decision to support the integration of different hardware platforms and operating systems. With the portable capabilities of Java and the standard of TCP/IP, we have hopes of building and testing a hybrid test system composed of many different operating systems connected and communicating together as a single system.

*3.1.2 Simulated Visualization.* This research effort did not initially involve modifying existing visualization software for rendering the NS-2 execution. However, to facilitate the receipt of data from NS-2 simulations and to test the ability to feedback commands through the mediator to NS-2, a visualization was required. To simplify the development effort of the visualization component of this effort, a simulated visualization (SimVis) is substituted for a fully functional visualization package. The SimVis is written in Java in a similar approach as the mediator to take advantage of code re-use where possible. The SimVis has one TCP/IP connection with the mediator to send commands and receive simulation data. The SimVis is a simplistic design to facilitate testing of full loop execution by allowing command

transmission with button pushes and windows to display the data received from the NS-2 simulation run and commands sent.

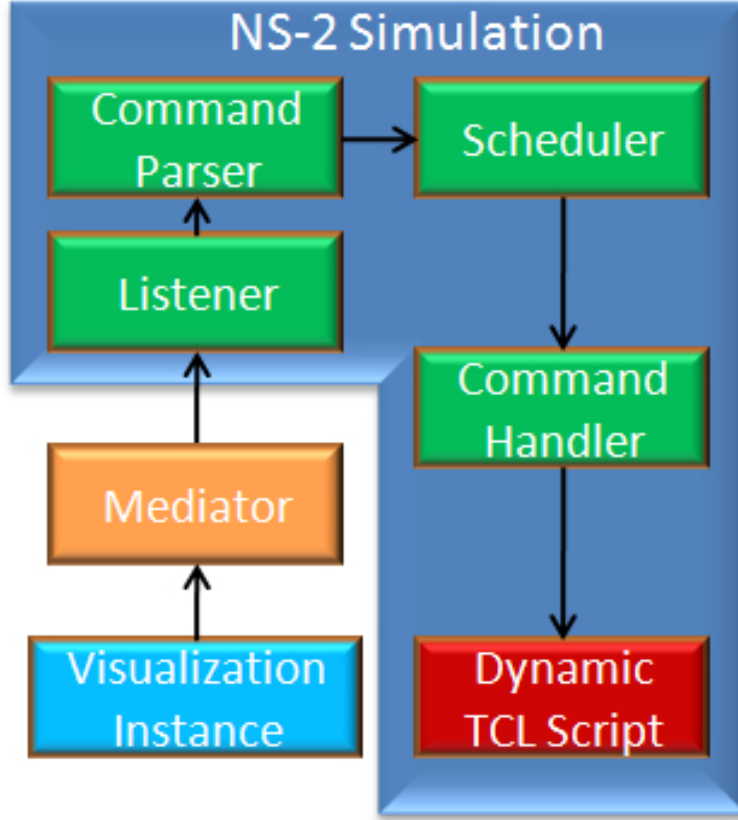


Figure 3.4: Illustration of command flow from visualization software through the mediator to NS-2. Once the command is received by the Listener class in NS-2 C++, it is passed to the CommandParser to schedule. The scheduler calls the command’s `handle()` method which creates a TCL script executed at the current simulation time. This in effect generates dynamic TCL that the simulation immediately executes.

*3.1.3 Command Reception and Handling.* NS-2 was chosen as the network simulator for this research. To implement a mediator with NS-2 we needed to extract data from NS-2 and replicate it for the visualization instances. Further, we needed a mechanism to inject the user generated commands back into NS-2. Our goal for implementing these changes was to minimize the changes and impact to the current execution of NS-2.

The goal of our design is to leave the original NS-2 execution untouched to preserve model data output. None of the changes or additions to the code effect the simulation data. The design allows the introduction of commands and schedules them at the top of the scheduler stack as the next event to be executed. Other events maintain their current scheduled times and use the original `handle()` methods without modifying the output data. This also has the benefit of allowing the use of previously written scripts with only minor modification (see Section 3.2.2.1).

The design consists of two new classes, the *Listener* and *CommandParser* and as well as a class for each *CommandHandler*. The Listener class is the main object for command reception and execution. Contained within the Listener object is the CommandParser object which in turn holds all CommandHandler objects. This design also implements separate execution from the main NS-2 thread. In addition, modified NS-2 schedulers were also developed to allow timing for realistic user interaction.

*3.1.4 Modified Simulation Execution.* Figure 3.4 highlights the execution path of a command from origination at the visualization end to the dynamic TCL script. A user provides command feedback through the capabilities of a linked visualization instance. These commands are sent to the mediator (and duplicated if necessary) which routes them to the network simulation(s). In this example, the NS-2 simulation receives the commands at the TCP/IP port contained within the Listener class. The Listener parses the received string and passes the information to the CommandParser which, in turn, determines the appropriate event(s) to schedule. The NS-2 scheduler then calls the appropriate CommandHandler `handle()` method. Each `handle()` method has access to the instance of TCL currently running. It passes new TCL strings up to the TCL instance, which executes the desired command effect (e.g., turning off a CBR traffic flow). This approach produces the same result as if the event was written in a hard coded TCL script prior to starting the simulation. This allows more flexibility and on-the-fly inputs for research or training purposes.

## 3.2 *Software Implementation*

As the software was implemented from the design, changes were made to improve each of the components discussed below. The Communication Mediator and Simulated Visualization were written in Java. NS-2 enhancements were written in C++ and script modifications were written in TCL. Each component implements the design decisions made above in Section 3.1 and software reuse was utilized when applicable.

*3.2.1 Communication Mediator.* The function of the mediator is to provide the connection of multiple simulations and visualizations. This role is sometimes known as that of a run-time infrastructure (RTI) [6]. Since the connections are TCP/IP sockets, the simulators and visualization instances can be co-located or at remote locations. The mediator is written in Java and takes advantage of the object oriented paradigm. Threading is utilized for socket listening as well as each connection establishment. Once a connection is established and the type of connection determined, it is added to a list of connections. The listing allows the mediator to duplicate data and/or commands for each simulator and visualization instance connected.

A single TCP/IP socket is used for the visualization instances, while two are required for the simulators. A single connection is efficient for visualizations since the client primarily listens for data which is a blocking socket operation. When a command is received from a visualization client, it can be cued for transmission when the next read operation occurs. The simulator sockets cannot operate in this manner. The first simulator socket, the *server* connection, is utilized to transmit data from the network simulator as it executes and a blocking read on that socket would disrupt the continuous data flow from the simulator. To alleviate this problem, the second socket, the *command* connection, was added to allow the first socket to exclusively send data. The *command* socket is utilized to transmit commands from

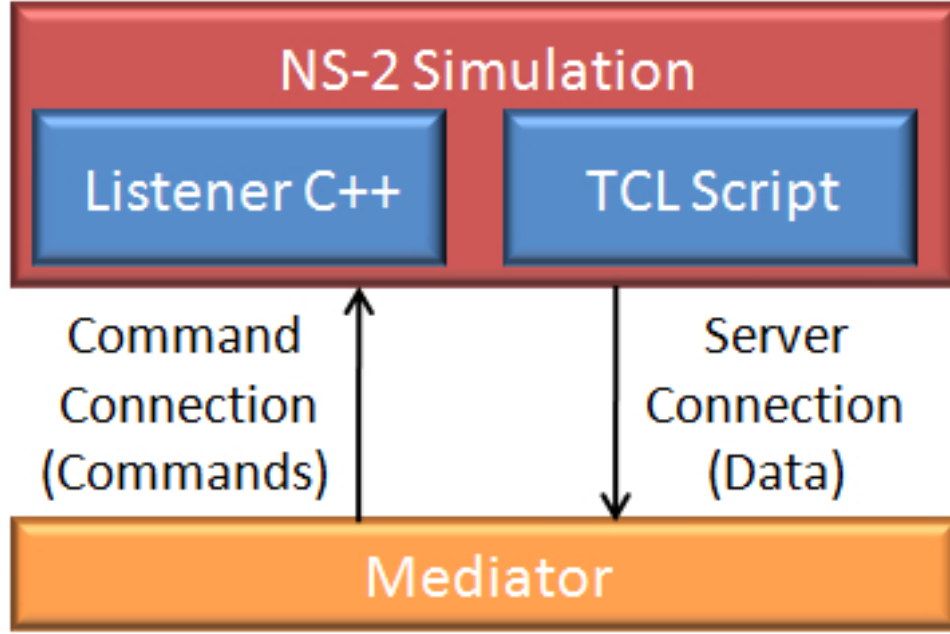


Figure 3.5: Illustration depicting the two required TCP/IP connections required between the mediator and a simulation (in this case NS-2). Commands are sent through the command connection and received by the Listener class in C++. The simulation data is transmitted from NS-2 to the mediator through the server connection created by the TCL script.

the mediator to the simulator. Figure 3.5 depicts the design for both sockets within the environment. Java socket code was developed using [17] and [15] as templates.

*3.2.1.1 Mediator Protocol.* The basic protocol established for the mediator allows TCP/IP to control the functionality. The objective was to keep the design simple without creating an entire large proprietary protocol language. Protocol strings are used in two instances: when a connection is established and when it is disconnected. When a connection is established, it is passed to a new thread for execution. The thread listens on the port for one of three connection protocol strings to determine what type of entity is establishing the connection. The three types of connections are **server**, **client**, and **command**. The server connection is established for simulation data transmission, while the client connection is configured for visualization instances. The command connection is for simulation command transmission after receipt from the visualization instance. One additional protocol

string, *Goodbye*, is used by the visualization or simulation to disconnect from the mediator.

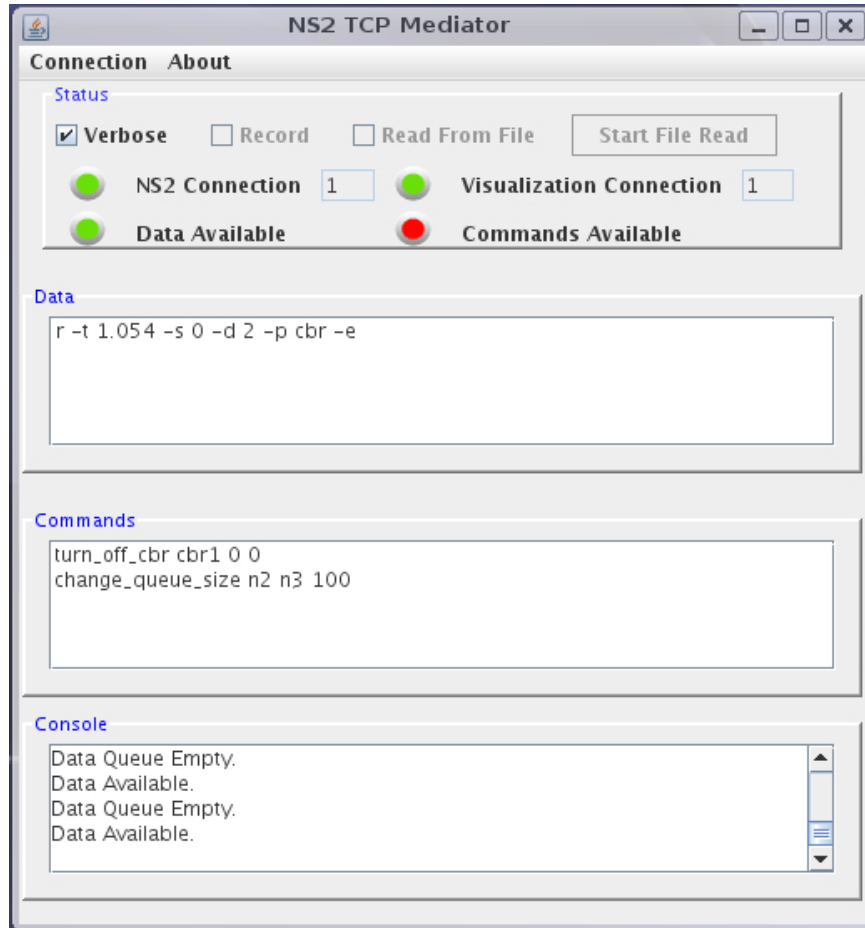


Figure 3.6: Mediator GUI with NS-2 connection, visualization connection, data, and command indicators. The connection indicators turn green when a simulator or visualization/s connect while the available indicators show the status of data and command transmission. The Data panel displays data from connected simulations on the way to visualizations. The Commands panel displays commands from the visualizations sent to the simulations. The Console panel displays mediator status messages.

*3.2.1.2 Graphical User Interface.* Figure 3.6 illustrates the mediator graphical user interface (GUI). The GUI status panel contains three check boxes and four red/green indicators. The *Verbose* check box enables display of data and commands for troubleshooting purposes. The *Record* check box creates a file to save the simulation trace for playback at a later time. The *Read* from file check box provides



the capability to playback a simulation trace file without a connection to a simulator utilizing a previously saved file. The indicator lights depict connections to the network simulator and visualization instance as well as data and command availability. The *Data* panel streams the data received from the network simulator or file while the *Commands* window displays the commands received from the visualization instances. The *Console* window reports status messages relating to the current state of the mediator application.

*3.2.2 Simulated Visualization.* The SimVis relied heavily on code developed for the mediator. The SimVis was developed to verify data receipt from a simulator through the mediator as well as issue commands to the simulation via the mediator. One TCP/IP socket is controlled by a separate thread and is primarily used to read incoming data. When the blocking read breaks to display the data, it checks to see if a command is queued. This approach upholds the primary use of the socket - data reception.

Figure 3.7 depicts the GUI of the SimVis. The *Status* panel contains two indicator lights for connection to the mediator and data receipt from the mediator as well as a button to allow connection to the mediator. The SimVis GUI's *Commands* panel holds five command buttons all utilized for testing the command handling capability of the NS-2 modifications. The buttons become active once a connection is established with the mediator. The *Data* panel displays data sent from the NS-2 simulation through the mediator and the *Console* panel displays messages relating to the current state of the SimVis program.

*3.2.2.1 Extracting Data from NS-2.* Traditionally, when a NS-2 simulation is run, the output is directed to one or more trace files. Instead of writing a trace file from the TCL script, our design utilizes a TCP/IP socket to send the simulation data to the visualization instance through the mediator. The trace is created in the same manner as usual, however, instead of writing the trace to a file, it is sent through the TCP/IP connection. If a trace file is still desired, the mediator has the

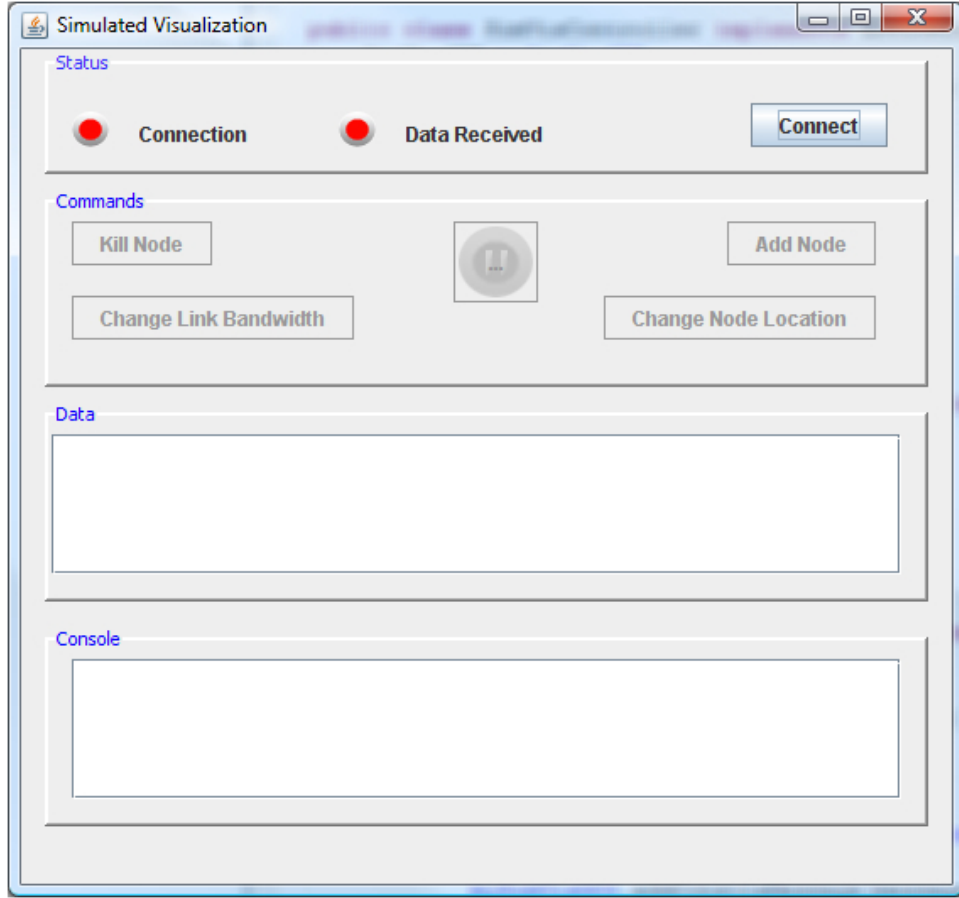


Figure 3.7: Simulated Visualization GUI with mediator connection and data received indicators. The connection indicator turns green when connected to the mediator. The data indicator turns green upon receipt of data from a simulation. The connect button connects to the mediator which enables the buttons on the Commands panel. Each command button has an assigned command it sends to effect the simulations model. The Data panel displays data received from a simulation while the Console panel displays SimVis status messages.

capability to write the file as it receives the data. Thus, previously written scripts can operate with only minor modification in order to redirect the trace output to a socket. The TCL socket script addition is depicted in Figure 3.8.

*3.2.2.2 Listener Class.* The Listener was the only dual TCL/C++ class developed. It creates a C++ TCP/IP socket and communicates with the mediator for command reception. The Listener is the gateway for command reception and

```

#Set up tcp connection
set server localhost
set $nf [socket server 1234]
fconfigure $nf -blocking 0
puts $nf "server"
...
# Tell the mediator to disconnect
puts $nf "Goodbye"

```

Figure 3.8: Example TCL Script for TCP/IP connection with the mediator enabling data transmission to the visualization.

execution. In order to allow separate execution of socket listening and NS-2 execution, a separate thread was required to perform all command translation and execution. The threading capability of the Boost library [4] was incorporated to perform this function. When the thread is created, it initiates a TCP/IP socket and connects to the mediator as a *command* connection. After the socket is initiated, the thread creates a singleton CommandParser object, discussed in detail below, to provide all command execution. Example code was found at [11] for C++ socket development.

```

#Instantiate a Listener object
set l0 [$ns listener]
...
#Commands and parameters for testing
$ns at 1.0 "$l0 readfile change_cbr_packetSize cbr0 2000 0"
$ns at 1.5 "$l0 readfile change_cbr_interval cbr0 0.25 0"
$ns at 3.5 "$l0 readfile turn_off_cbr cbr1 0 0"

```

Figure 3.9: Example TCL script for CommandHandler development depicting the code required to test the CommandHandler code without running a visualization instance.

Another key benefit of the Listener is that since it is a TCL object, it can be instantiated in a TCL script and commands can be scheduled in the same script using this object. This ability becomes useful when testing new commands, allowing command execution without coupling with the mediator or a visualization suite. TCL commands also do not require a recompilation of the C++ code saving precious

research time. Figure 3.9 is an example of a TCL script for command handling development.

The listener also contains the IP and port for the TCP/IP connection. the default values are: IP *localhost* and port *1234*. If the mediator is located on a remote machine, the IP and port must be configured to that machines IP and the port associated with the mediator. NS-2 must be recompiled after this change for it to take affect.

*3.2.2.3 CommandParser Class.* The *CommandParser* class was created to link the *Boost::thread* with NS-2's scheduler. The *CommandParser* has the ability to find the current TCL instance and utilize it to communicate with the NS-2 scheduler. Upon initialization, *CommandParser* creates a singleton instance of each defined command handler. Once selected, the handler for that event is scheduled with the NS-2 scheduler.

The *CommandParser* uses a series of *if / else* statements to determine which command was passed from the listener to it's *parse()* method. Once the appropriate command has been matched, the singleton instance of the handler is scheduled as a new event. Some commands may require scheduling multiple events with each with their own handler. This section of the code may benefit from lazy initialization and is discussed in detail in Section 5.3.2.

*3.2.2.4 CommandHandler Class.* The *CommandHandler* class was designed as an abstract class to simplify the addition of increased command capability. The abstract class handles the overhead of the *Handler* class from NS-2 as well as specific functionality and variables for command execution. This approach means a user is only required to write the handle routine for command execution. Thus, the mechanisms and required activities are already complete, a user needs only to provide the command specific activity for the individual new command they want to add interaction for in the simulator.

Each of the CommandHandlers written for our research utilize this method and contain only a `handle()` method. The commands written to date are primarily for wired applications, however, the same approach will work for wireless commands as well. The `handle()` method code is a simple TCL script that is passed up to the instance of TCL currently running. The NS-2 scheduler calls the handle method for the event that was scheduled by the CommandParser just as if it came from the TCL script file. Thus, the NS-2 TCL script is amended to include the code required to execute the remote command sent by the user from the visualization instance. Commands can utilize more than one handler as in the case of the `change_cbr_interval` command as shown in Figure 3.9. This particular command requires three event handlers: stop CBR, change CBR interval, and start CBR.

A detailed command development is outlined in Section E. This section contains the specific files containing the command code as well as a description of the naming conventions and inheritance to use when developing new commands. Basic command testing is also outlined in this section to facilitate successful enhancements.

### ***3.3 Additional NS-2 Enhancements***

In order to facilitate the incorporation of NetVis, additional NS-2 enhancements were required. Usability drove most of these changes, due to the speed of NS-2 execution. The simulator, when executing a small simulation, finishes a 30 second, four node simulation within a few seconds. It is problematic when the user is trying to interact with the simulation after it has already completed execution. Two paths were developed to attack this problem and allow NetVis to perform command feedback with NS-2. Both enhancements are new schedulers that inherit from the calendar scheduler in NS-2.

*3.3.1 Clock Time Scheduler.* Upon initial testing of the command reception code with a four node scenario, it was discovered that NS-2 simulations completed before commands could be handled. The events were scheduled and executed, but

the simulation time had already finished, so the activity had no effect. The problem was NS-2 ran faster than the human interaction from the visualization.

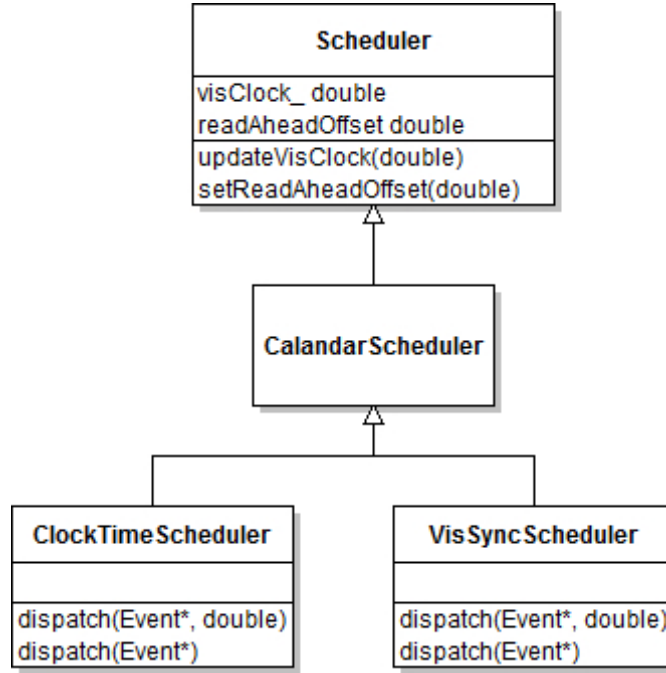


Figure 3.10: Scheduler design concept illustrating inheritance for both the ClockTimeScheduler and the VisSyncScheduler. The parent scheduler incorporates two variables, `visClock_` which holds the last clock time received from the visualization and `readAheadOffset_` which keeps NS-2 ahead of the visualization. The parent scheduler also includes two additional methods to update the previous variable values. The ClockTimeScheduler utilizes a modified `dispatch(Event*, double)` method. The VisSyncScheduler also utilizes a modified modified `dispatch(Event*, double)` method that relies on the two new parent variables.

To mitigate this problem, a new NS-2 scheduler was designed that can help throttle down to a user friendly speed. The ClockTimeScheduler is our implementation for this problem and attempts to run NS-2 simulations in near-clock time. Figure 3.10 highlights the software engineering approach taken to develop this scheduler. The scheduler inherits from the Calendar scheduler contained in NS-2. The only method modified in this application is the `dispatch()` method. To facilitate the overloading of this method, the `dispatch()` method in the scheduler class was changed to virtual void. No additional variables nor methods were affected by this design approach.

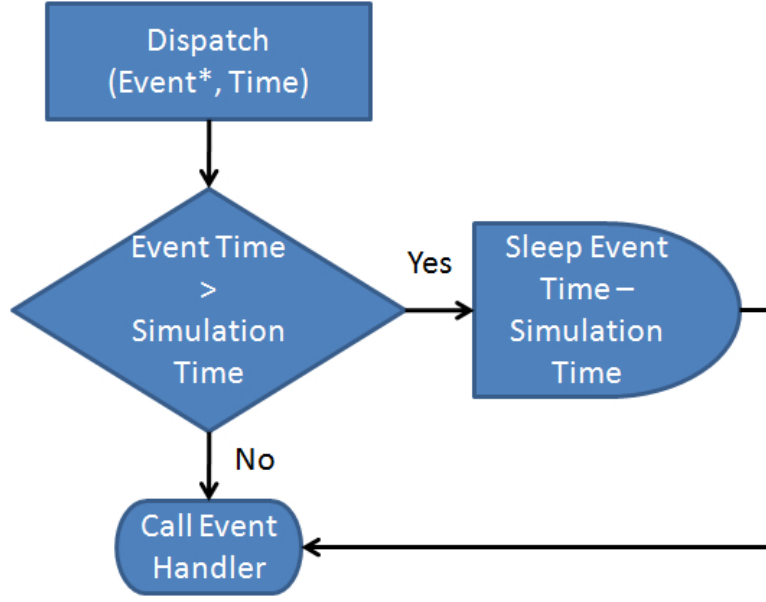


Figure 3.11: Dispatch logic incorporated within the ClockTimeScheduler. If the next event’s time is greater than the current simulation time, the scheduler sleeps the difference between the two times. After sleeping, the event handler is called. This allows the scheduler to approach clock time execution.

Figure 3.11 depicts that as the scheduler pulls an event off the stack and prepares to dispatch it, it calculates the difference between the current simulation time and the next event’s scheduled time. The scheduler then delays (sleeps) for the difference in time between events. This delay approximately traces wall clock time to effectively slow simulation execution and allow command feedback. This new scheduler is applicable with a visualization that can render the simulation to a screen in clock time. This solution will not be as effective if the visualization software runs slower than clock time. For example, if the number of rendered objects is high, the rendering will be slowed below clock time. Thus, the ClockTime scheduler may have limited application potential.

*3.3.2 Synchronized Scheduler.* To solve the problem of linking a visualization that runs slower than clock time (which is the case with NetVis for large complex networks), another scheduling approach must be taken. If the macro problem is eval-

uated closer, there are times when network events are little or non-existent and other times where network events are abundant. These two modes of visualization are quite different when it comes to rendering time. As the network becomes more complex and network events increase, the rendering time increases. Thus, the execution and rendering of a visualization to a screen is not a static time constraint, but very dynamic. The best method to control a time dynamic execution is to use a synchronized approach.

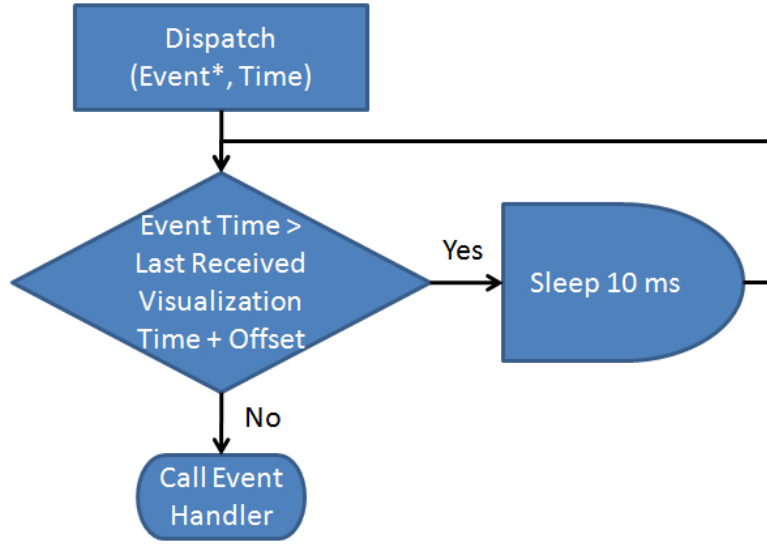


Figure 3.12: Dispatch logic incorporated within the VisSyncScheduler. If the next event’s time is greater than the last time received from the visualization plus the offset, the scheduler sleeps for 10 milliseconds. After sleeping, the scheduler reevaluates the logic. When appropriate, the event handler is called. This allows the scheduler to approach a synchronized execution with the visualization.

To solve the problem utilizing a synchronization approach, the VisSync scheduler was developed. As shown in Figure 3.10, this scheduler also inherits from the Calendar scheduler available in NS-2. Two additional variables are added to the main scheduler to facilitate this approach; `visClock_` which keeps track of a “heartbeat” (time) sent from the visualization and `readAheadOffset_` which is a definable time used to keep the execution of the simulation ahead of the visualization. Two new methods were also added to the main NS-2 scheduler code, `updateVisClock(double)`



and `setReadAheadOffset(double)`. `updateVisClock` updates `visClock_` with the latest time hack sent from the visualization while `setReadAheadOffset` updates the offset and is also sent from the visualization. Both of these commands are not scheduled, but updated directly from the `CommandParser`. These additional variables and commands have no effect on existing schedulers or user defined schedulers, for when called, the variable is updated and is not used by any other scheduler code except for the `VisSync` scheduler. The changes in the `VisSync` scheduler focused on the same method as with the `ClockTime` scheduler, the `dispatch()` method. This method was modified to evaluate the difference between the current simulation time and the last clock “heartbeat” received from the visualization to determine if the event should be dispatched. If the difference was greater than the `readAheadOffset_`, the event would not be dispatched and the scheduler would sleep for ten microseconds then return to evaluate the event once again. Figure 3.12 depicts the logic used in this approach.

The development of this additional scheduler also provided a beneficial side effect when coupled with `NetVis`. If one instance of `NetVis` is utilized in a simulation system with the `VisSync` scheduler, `NetVis` has the ability to pause and restart the simulation. This could have many benefits in an educational environment to pause the simulation and explain what is occurring before continuing on with the simulation. The pause has no effect on the outcome of the simulation, other than it stops the execution until play is pressed.

### ***3.4 NetVis Enhancements***

In order to better test the functionality of command feedback, an enhanced visualization software package was required. `SimVis` provided the basic functionality of receiving data and transmitting commands, but did not render the network or events. In a parallel effort `NetVis` has been enhanced to include command feedback capability. This provides an end-to-end system and demonstrates the power of user interaction with a simulation while it executes. In addition to command feedback,

clock synchronization functionality was also added to make the complete system more user friendly.

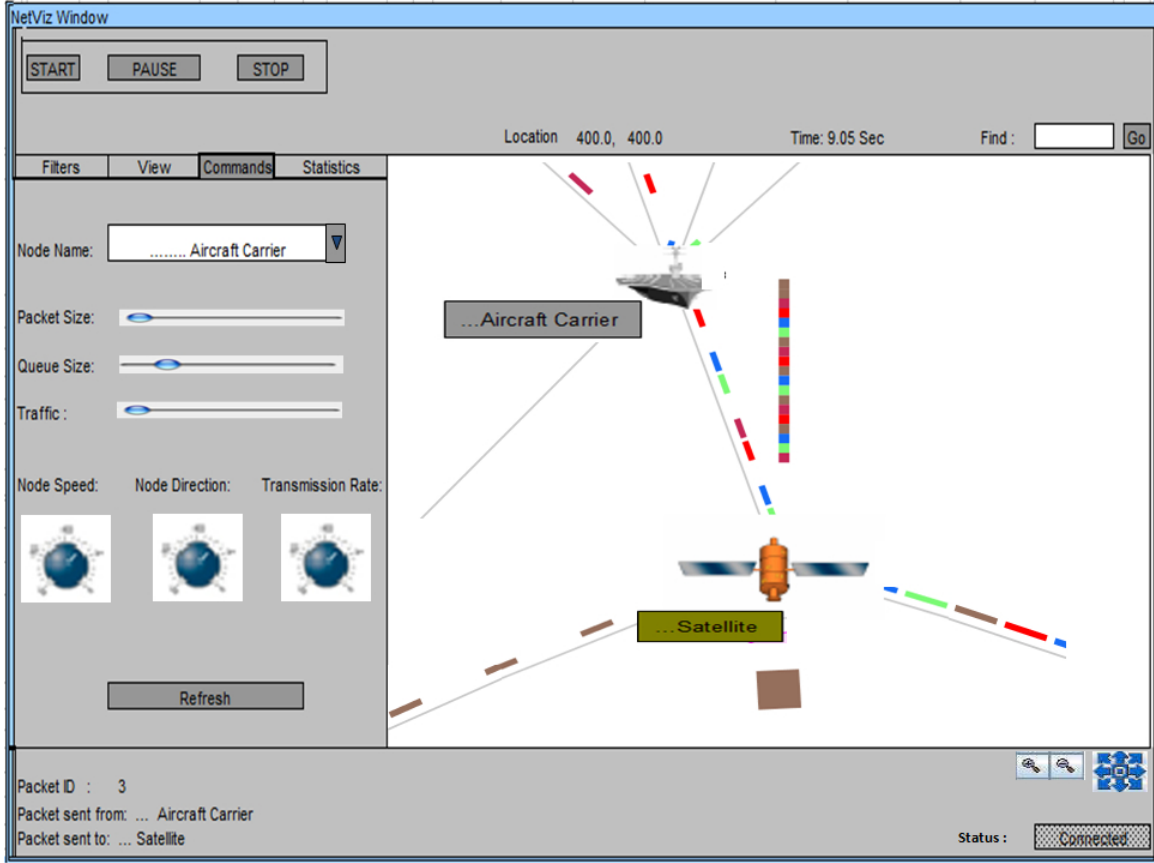


Figure 3.13: Image of Image of NetVis depicting the newly designed command panels. Packet size and queue size can be dynamically changed by adjusting the appropriate slider bar. This utilizes the new command feedback capability of the modified NS-2 used in this research.

*3.4.1 Command Feedback.* To support the command feedback capabilities introduced into NS-2, NetVis was enhanced as a parallel research effort. User control panels were added to the GUI to provide the capability to select a link or node and change the parameters of the item through command feedback [28]. Figure 3.13 depicts the user's control panel allowing command feedback to NS-2 through slider controls for packet size and queue size. These commands required code development within NS-2 to take the desired actions through dynamic TCL creation. Additional

command capability is currently in development for NetVis implementation. These enhancements will improve the usability of the command feedback system as a whole.

Wired and wireless commands require unique tailoring as well as scenario specific commands. Some basic commands can be re-used or used as a component of a complex commands. The effort to produce additional commands is a joint effort between the NetVis group and this research effort.

*3.4.2 Clock Synchronization.* When running a complex network with substantial network traffic, NetVis does not render the simulation visualization in wall clock time. After the simulation starts, it could take NetVis many seconds to display one second of simulation time. Using the `ClockTimeScheduler` was not sufficient to solve the problem since the visualization could severely lag the simulation software execution. This drove the design of an additional scheduler, `VisSyncScheduler` to alleviate the problem. NetVis was modified to send a clock pulse command `update_vis_clock` once a second with an offset of 1.1 seconds to approach a synchronized co-simulation. The details of the command handling in NS-2 are described in Section 3.2.2.4. NS-2 holds simulation execution until each clock update received from NetVis see Section 3.3.2 above.

In addition to the `update_vis_clock` command, an offset is used to ensure that NS-2 execution stays ahead of NetVis. The command for sending an offset is `set_read_ahead_offset`. The offset is tailored for each individual TCL script and explained in more detail in Section 4.2.5.2. NetVis sends the offset when the start button is pressed to provide a way to start a synchronized simulation. These modifications have greatly increase usability of the command feedback system.

*3.4.3 Co-Simulation with SimVis.* One of the benefits of simulation with the mediator is the ability for co-simulation. NetVis has the ability to render the network events and provide command feedback, but new untested commands require software additions to the visualization code. A simplified way to add capability for

additional commands is to run SimVis in parallel with NetVis. Both clients have the capability to send commands and NetVis displays the simulation. New commands can easily be tested using SimVis while watching for the results on NetVis. The mediator facilitates the connection of multiple visualizations and does not limit the number of any specific type. This allows the possibility for an array of visualization clients to run concurrently. NetVis, SimVis, and potentially other visualization suites can be adapted for use.

## IV. Improving Performance and Usability

Testing and evaluation of the system is outlined in this section, as well as improvements made to the components of the system during and after testing. Testing was accomplished in an incremental method and needs for additional functionality, usability, and efficiency were addressed.

### 4.1 *Additional Enhancements*

Three areas of the command feedback simulation system were enhanced during the testing phase. The first, is a TCL script addition improved the socket creation time. The next enhancement was inserting two additional parameter slots into the command string parser allowing complex wireless command development. The last development was the morphing of SimVis into the Configurable Command Tool (CCT) providing additional capability and flexibility.

```
#Set up tcp connection
set server localhost
set $nf [socket server 1234]
fconfigure $nf -blocking 0
puts $nf "server"
#Immediately set up connection
$nf flush
...
# Tell the mediator to disconnect
puts $nf "Goodbye"
```

Figure 4.1: Example TCL script for TCP/IP connection with the mediator enabling data transmission to the visualization. Note the additional line “flush” to facilitate immediate connection with the mediator

*4.1.1 TCL Script Enhancement.* During testing with a simple four node wireless scenario, the data TCP/IP socket between NS-2 and the mediator was not connecting until 55 seconds into the simulation. Initially it was thought to be a lack of data transmission at the beginning of the scenario, so a modified script was written with early data transmission. This did reduce the delay to 17 seconds, but it was still

an unacceptable situation, for commands could not be sent during that initial period of time before the socket connection occurred. The final solution was to add one line to the TCL script to flush the socket after ‘‘**server**’’ was sent to the mediator to initialize the connection. This change allowed an immediate TCP/IP connection before the scenario began. The modified code is outlined in Figure 4.1.

*4.1.2 NS-2 Enhancement.* As wireless scenarios were generated for testing purposes, it was discovered that three variables passed with a command string would not be sufficient. To move a wireless node, an X and Y coordinate are required as well as a speed variable. After evaluation of most possible commands, a design decision was made to increase the number of variables to five. This does not include the command itself and should account for all existing command possibilities. The possibility for a shortfall still exists for user defined commands that would require in excess of five parameters.

*4.1.3 Tool Enhancement.* As the design was constructed and testing progressed, additional potential improvements surfaced. The SimVis was initially envisioned to fulfill intermediate testing until a visualization software package was modified for full interaction capability. Rather than throw the tool away, the software was modified for additional usability and renamed the Configurable Command Tool.

```
cbr off
turn_off_cbr cbr1 0 0
move cbr
move_cbr cbr1 udp2 0
heartbeat
update_vis_clock
change interval
change_cbr_interval cbr0 0.25 0
```

Figure 4.2: Example configuration file for Configurable Command Tool . The first line defines the GUI button text while the second line defines the command sent when the button is pressed. This example code defines four of the six available buttons.

*4.1.4 Configurable Command Tool.* The SimVis software was evaluated to determine if it could have beneficial use beyond initial system testing. With minor modifications to the Java code, the SimVis has evolved into the Configurable Command Tool (CCT). The CCT uses the same functionality as SimVis by connecting to the mediator to facilitate simulation data receipt and command transmission. The main difference with the CCT is it now reads a configuration text file that allows dynamic creation of up to six command buttons. An example configuration file is shown in Figure 4.2. Two lines define each button; the first defines the button text and the second defines the button command. If a line consists of only a carriage return the button will not be defined. Also, if less than twelve lines are in the configuration file, some buttons will not be defined. The non-defined buttons will not be enabled during connection with the mediator. Figure 4.3 illustrates the CCT GUI. The GUI has two additional buttons for a total of six configurable items.

The CCT is envisioned for use by a researcher that requires the capability to send commands without modifying visualization software. The CCT can be used in parallel with a visualization, in our research, it was used in conjunction with NetVis. This allows the researcher to watch the visualization for changes that occurred after command feedback issued from the CCT.

Another direct application for the CCT is for instructor use in an educational environment. Figure 4.4 revisits the educational application of our system with the CCT in use. This configuration allows the instructor to predefine commands to evoke during simulation execution creating situations the students react to. The instructor can also have a visualization running in parallel on his/her terminal to view the effect of the commands and student interactions.

An additional change to the SimVis functionality in the CCT is the timing associated with sending commands. After the VisSync scheduler was written, a problem was discovered where a command could not be sent after the initial pause of the simulation (while it waited for a heartbeat). This included the inability to send an

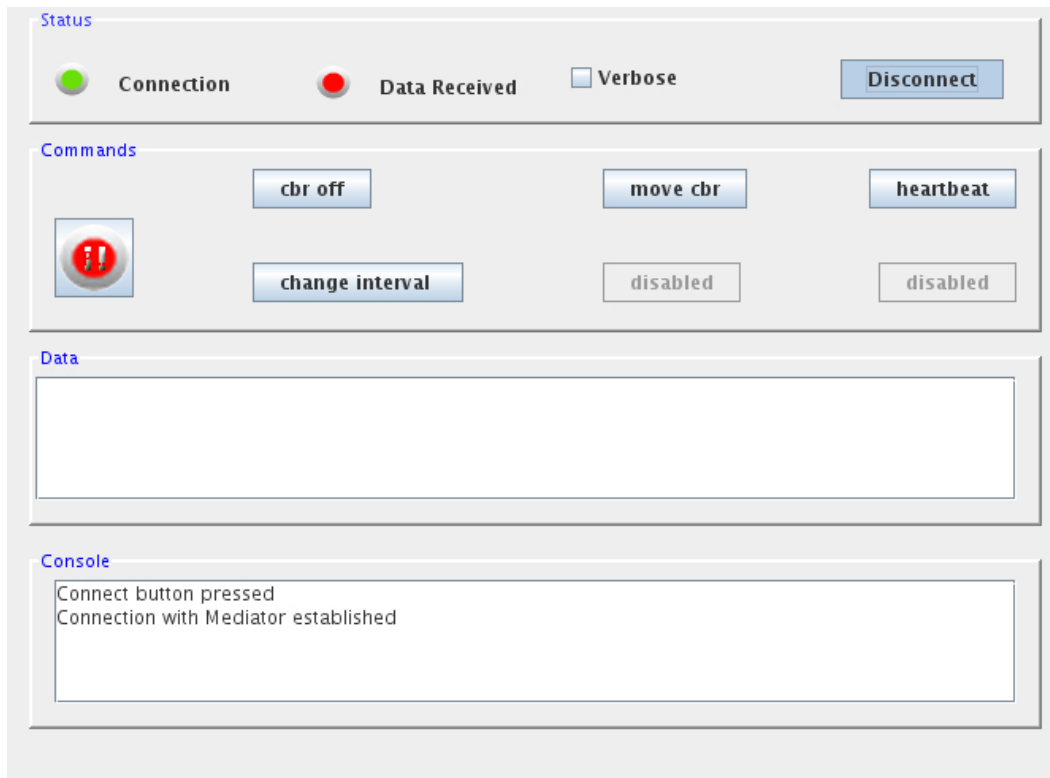


Figure 4.3: Command Configuration Tool GUI illustration the six definable command buttons. A configuration file is utilized to define the button labels and commands sent when pushed. Buttons not defined are disabled during program execution. The buttons defined above are “cbr off” - used to turn off cbr1 when pressed; “move cbr” attaches cbr0 to node 2; “heartbeat” sends the command to update the scheduler visClock; and “change interval” which changes cbr1’s interval to 0.25 seconds. All these button definitions were used to test their relevant commands during development.

updated visualization time which caused the system to deadlock. The code in the CCT was evaluated and a timeout was added to the socket to allow the CCT to break from it’s blocking read to check for a command in the queue. After this change was implemented the CCT was able to interact correctly with the new VisSync scheduler.

Another improvement in the CCT is the ability to facilitate a connection to a remote mediator. When the connect button is depressed, a verification window appears asking if IP *localhost* and port *1234*. If the mediator is not running on the same machine, the appropriate IP and port can be entered. This change allows remote connection without code changes.



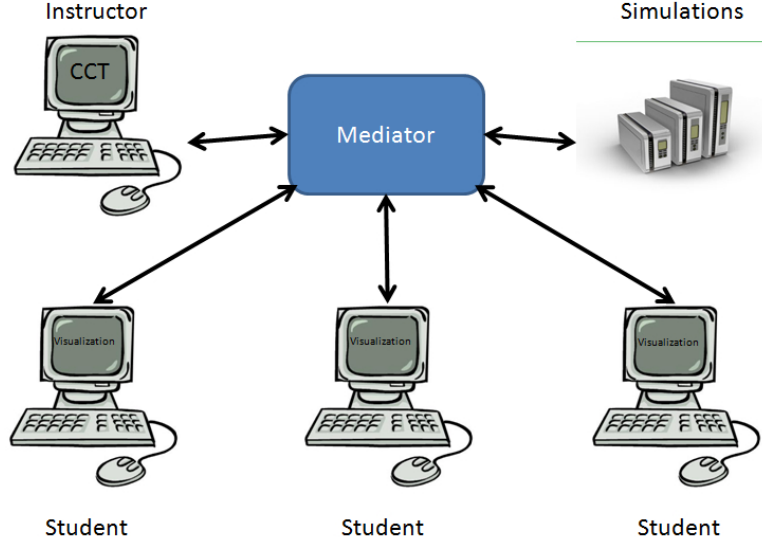


Figure 4.4: Illustration of an educational application of the system using the Configurable Command Tool for instruction. The instructor can predefine commands to invoke events and wait for student reactions.

## 4.2 Testing and Evaluation

In order to successfully test a command feedback simulation system, a full system test is required from the simulation to the visualization. Taking a systems engineering approach, the complete system is component tested, then gradually built into larger sub-systems, and finally a complete system. Testing this research has been divided into subsystem testing, starting from smaller modules, and building up to a complete system test to follow the systems engineering approach. This modular testing is beneficial not only to validate the system, but also to develop additional commands in the future.

Testing our design can be accomplished without employing a visualization instance. First, the individual commands can be tested within the NS-2 simulator environment as scripted TCL commands. After the commands are finalized and tested as scripts, we introduce a simulated visualization (SimVis) into the system. SimVis brings the mediator into the operation for full loop testing. After successful testing with SimVis, a modified NetViz software package was utilized in testing to enable visual feedback of command reception.

To accomplish this testing, a linux compatible computer will be required capable of running NS-2 version 2.32. It will also require Java Virtual Machine version 1.6.0 or greater. Software components required are the modified NS-2 code, the mediator, and the SimVis software modules. Example TCL scripts from [13] will be used to develop scenarios. Initial testing will be accomplished on a single personal computer.

In addition to the testing described above, full system testing will also be accomplished in a hybrid environment as well as on a mock-up educational system. The hybrid system will be composed of as many different hardware platforms and operating systems as feasible. The educational system will emulate a classroom environment with instructor and multiple student computers.

*4.2.1 Mediator Testing.* Initially, the first component developed was the mediator. Thus, testing began with initial connections to the mediator to establish the system communication framework. Successful exit criteria is to verify that the mediator could support connection of multiple simulators and visualizations as well as receive data from all connected components.

The first test accomplished was to link small Java programs to simulate a simulator (server) and a visualization (client) to the mediator with the TCP/IP connections. These simulated modules sent a few lines of data. The mediator has indicators for NS-2 and a visualization that are initially are red but turn green when a connection occurs. These were observed for a state change to verify a connection was established. In addition to the indicator lights, the mediator has a verbose check box. When checked, the mediator will display data received from a simulator and commands received from a visualization. These commands are removed after they are sent to the visualization and simulator respectively. This feature facilitated the verification of test data sent from the simulated software modules.

Once the initial testing was accomplished with one of each simulated client and server component, a simple test was conducted to verify that multiple servers and clients could connect. One mediator was instantiated while three servers and three

clients were run. Each component sent a different set of data in order to verify which component the data came from when displayed by the mediator. Again, the mediator was successful in connecting to and displaying the data from each component.

*4.2.2 Testing the NS-2 Data Connection.* Once the mediator was initially tested to support connections and facilitate data communication, the next step was to introduce real simulator data. As described in Section 3.2.1 a socket to stream data from NS-2 is constructed in a TCL script instead of writing it to a trace file. By utilizing the data socket in a TCL script, the ability to test a simulator instance with the mediator was realized. Successful exit criteria for this testing is a TCP/IP connection from a NS-2 simulation to the mediator as well as transmission of a complete set of simulation data.

This testing was successfully demonstrated on a single computer. The NS-2 connection indicator turned green to verify a socket connection was established. With the verbose button checked, the data was displayed in the mediator's *Data* window. This proved that the data was received by the mediator, but did not verify that the data was complete.

To test the totality of the data, another test was accomplished. The mediators' record check box was selected to allow the mediator to write to a file. A basic TCL script was run first *without* the TCL socket connection writing the simulation to a file. The same TCL scenario was run, this time *with* the TCL socket employed to stream the data to the mediator. Since the mediator had record checked, it wrote the data to a file. Once both simulations were complete, the Unix command `diff` was utilized to determine if the simulation data was different. No differences were noted, proving that the mediator was receiving the complete set of simulation data.

*4.2.3 Testing Simulation Data with SimVis.* Once NS-2 had successfully connected and passed data to the mediator, the next logical step was to verify the mediator would transparently (without modification) pass the data to a visualization

and multiple visualizations. Once SimVis was developed, this testing became possible. Another capability SimVis has is command transmission. Even though command reception was not yet a reality with NS-2, the capability through the mediator could be verified as simulation data was being received from NS-2. Successful exit criteria from this testing phase is successful data receipt by SimVis from a running NS-2 simulation through the mediator. The other exit criteria is successful receipt of command strings by the mediator from SimVis as simulation data is streaming.

This testing was performed on a single PC running a Linux environment. Initially one instance of SimVis was employed to perform testing. The mediator was started, then SimVis, and finally the NS-2 simulation. The mediator indicator lights verified connection from both SimVis and NS-2. Data was verified at the mediator via the verbose check box. Data receipt was verified at SimVis in the *Data* window. As the simulation data streamed to SimVis, command buttons were depressed to transmit command strings to the mediator. The command strings were successfully displayed on the mediator's *Commands* window. This verified that the commands could be sent at the same time data was received by SimVis.

An additional test was performed with a slight modification to the SimVis code to write the data to a file after receipt. This allowed the same `diff` check as performed with the mediator. The file was compared to the same simulation file produced by NS-2 without a socket connection. No differences were noted.

With the success of a single instance of SimVis, another test was performed with three instances of SimVis connected to the mediator. Each of the three instances received the same simulation data, verifying that the mediator successfully multiplexes the simulation data to multiple visualizations.

*4.2.4 Command Testing.* Now that data flow had been tested through the entire path and command receipt was successful at the mediator, command reception required testing. Command reception and handling is in essence a subsystem of it's own. As such, it can support component testing as well as subsystem testing. The first

```
#Instantiate a Listener object
set l0 [$ns listener]
...
#Commands and parameters for testing
$ns at 1.0 "$l0 readfile change_cbr_packetSize cbr0 2000 0"
$ns at 1.5 "$l0 readfile change_cbr_interval cbr0 0.25 0"
$ns at 3.5 "$l0 readfile turn_off_cbr cbr1 0 0"
```

Figure 4.5: Example TCL Script for CommandHandler Development depicting the code required to test the CommandHandler code without running a visualization instance.

test of command handling was exercising the Listener class. Successful exit criteria for this test would be a simulation event generated by the listener class in a tracefile.

By utilizing the Listener class in a TCL script, the command execution was tested utilizing only NS-2. This method does not send the command over a TCP/IP connection, nor does it require the mediator. The development of specific command handling code can be tested within a TCL script by instantiating a Listener and passing commands to that Listener object. This greatly eases command handler development. The second line of Figure 4.5 contains the command “`change_cbr_packetSize cbr0 2000 0`”. This provides the CommandParser with the information that the command is `change_cbr_packetSize` with parameters `cbr0` (change effects `cbr0`), 2000 (new packet size), and 0 (this is an unused parameter for this specific command).

The TCL script was run without a TCL socket connection and no additional components (i.e. mediator or SimVis). A NAM tracefile was created from this simulation and viewed using NetVis. NetVis allowed us to visually detect if the desired outcome was executed successfully. When the TCL script shown in Figure 4.5, NetVis displayed a change in `cbr0`’s packet size at 1.0 seconds into the simulation as well as a change in `cbr0`’s interval at 1.5 seconds and `cbr1` turned off at 3.5 seconds into the simulation. This validated that the commands were correctly written and could be invoked by the Listener class.

*4.2.5 Testing with NetVis.* After NetVis was modified to support connection with the mediator, it was brought into system testing. Initially, NetVis was tested in the system and results could not be obtained. The execution of the simulation was too quick for human interaction. To facilitate further testing, two additional NS-2 schedulers were developed to support NetVis' usefulness in the command feedback simulation system. Each of these schedulers required testing to prove their correct functionality. The test system included NS-2 as the simulator, the mediator, the CCT, and NetVis as the visualization package. This testing was accomplished on a single computer within a Linux environment.

*4.2.5.1 Testing with the ClockTime Scheduler.* Development of the ClockTime scheduler was out of necessity to slow down the execution time of NS-2 to allow NetVis to render the simulation and still allow time for command feedback (see Section 3.3.1 for additional details). To invoke the new scheduler only requires the addition of one line in the TCL script as depicted in Figure 4.6. Successful exit criteria for this test is the visualization clock should closely approximate wall clock time. An additional factor would be successful command feedback from NetVis.

NetVis was able to render a simple wired scenario consisting of four nodes with a minor amount of network traffic. The scheduler worked as required, slowing the simulation execution to approximately wall clock time. The positive effect of this was that we actually had time to interact and test command feedback to the simulation. A downfall to this scheduler was that NetVis did not render the simulation in wall clock time. This meant that the simulation execution had to be slowed even further.

```
#Use the ClockTime Scheduler
$ns use-scheduler ClockTime
...
#Use the VisSync Scheduler
$ns use-scheduler VisSync
...
```

Figure 4.6: Example TCL Script for invoking a ClockTime or VisSync scheduler.

This approach did have an initial benefit with simple wired scenarios, but with complex wired and wireless scenarios the render time of NetVis was still much slower than wall clock time. In fact, it was slower by a factor of 10 or more. In a simple wireless scenario with three nodes, the time for NetVis to start rendering, issue a command, and respond to the simulation change was 100 seconds. Another alternative was to develop a way to synchronize NetVis with NS2 to produce a more interactive system. This requirement drove the design of the VisSync scheduler.

*4.2.5.2 Testing with the VisSync Scheduler.* The VisSync scheduler was implemented as a necessary enhancement to the NS-2 framework to allow true user interaction with a simulation as it executed. The execution speed is driven by a “heartbeat” issued by the visualization and an offset to ensure NS-2 stays ahead of the visualization. This additional scheduler allowed the testing of complex wired as well as wireless scenarios without the large lag encountered with the ClockTime scheduler. Section 3.3.2 explains this scheduler in detail. Successful exit criteria for this test would be that the simulation execution would parallel the visualization time clock, staying ahead by a specified time offset. This should improve the ability to interact with the simulation.

Testing was conducted as with the ClockTime scheduler. Results were very positive. The simulation execution would run for the amount of time specified by the offset. At this point it would pause and wait for NetVis to send it another time hack at which time it would continue execution to the the time plus the offset. This provided the capability to indefinitely pause the execution by pausing NetVis. Execution would continue once “play” was pressed. Command feedback was extremely effective with this scheduler and additional details are stated below in Scenario Testing.

*4.2.6 Scenario Testing.* Different scenarios can be generated for NS-2 applications. Wired scenarios are usually fixed position nodes, while wireless scenarios have the capability for node mobility. These generalized scenarios can also be combined for hybrid network simulation. Scenarios are written in TCL scripting language

and do not require running a compiler. Changes can be made and immediately run to test the scenario.

Scripts utilized in testing were primarily from Marc Greis' *Tutorial for the Network Simulator "ns"* [13]. These scripts were modified to include the TCL TCP/IP socket and additional schedulers as required. Scenario testing started with the examples as-is, containing all the priori scripted events. As testing progressed the scripts were modified to generic scenarios lacking these scripted events during the execution (i.e. cbrs turned off or wireless nodes moving at time x). This produced a stepping stone approach similar to Section 4.2.4 with a baseline simulation and allowed the recognition of command handling without distraction from TCL script timed events.

Initial scenario testing was performed on a single computer running a Linux environment. All software components were instantiated on this computer and consisted of the enhanced NS-2 simulator, the mediator, the CCT for command invoking, and NetVis for visualization of the network events.

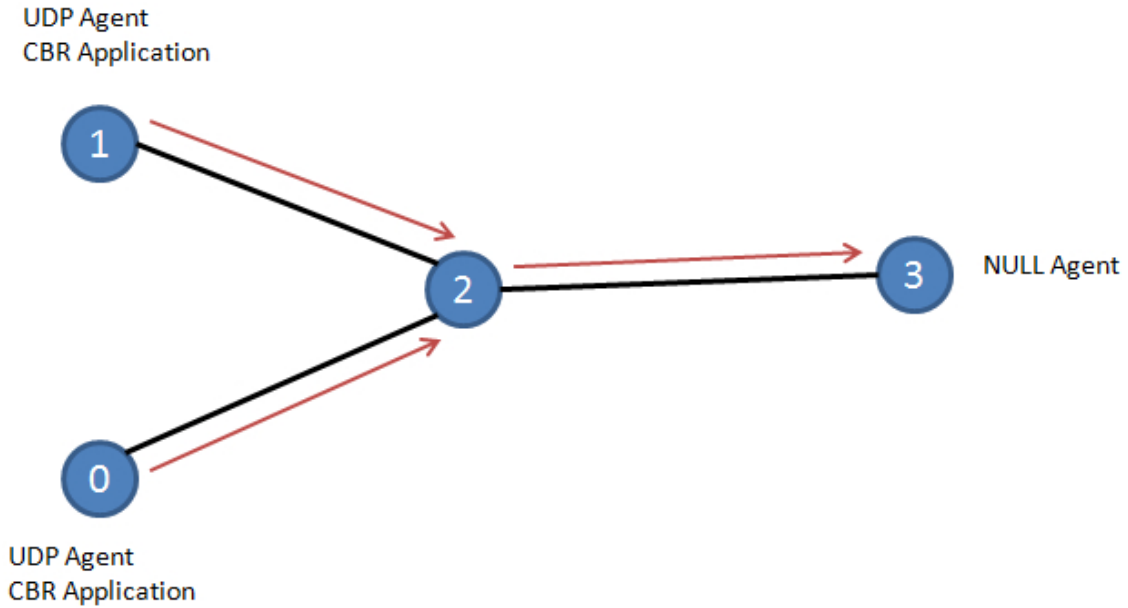


Figure 4.7: Wired TCL script used for testing depicting nodes zero and one with UDP agents and CBR applications. Nodes zero and one generate constant traffic through node two to reach the NULL agent at node three.



*4.2.6.1 Testing Wired Scenarios.* Wired scenario command generation and testing consisted of tutorial V *Making it more interesting* [13]. This scenario is a four node configuration as depicted in Figure 4.7. Traffic is generated at node zero and node one by UDP agents and cbr applications. This network traffic is sent through the network links to node two which relays the packets to node three's null agent. To facilitate testing of the command feedback code, the cbr applications start sending data at the beginning of the simulation and don't stop until the end of the simulation. This provides the full simulation time for command feedback testing. The original simulation time for this example scenario was five seconds. Due to the limited execution time, the length of the simulation was also increased to 45 seconds to allow adequate testing.

Specific commands tested were: `turn_on_cbr`, `turn_off_cbr`, `move_cbr`, `change_queue_size`, `change_cbr_packetSize`, and `change_cbr_interval`. Each command was issued from the CCT and the change viewed on NetVis. `turn_on_cbr` and `turn_off_cbr` start traffic generation and end traffic generation respectively and each command requires one parameter (for the cbr in question). `move_cbr` moves the cbr application from one agent to another and requires two parameters for the agents to move the cbr from and to. `change_queue_size` is utilized to change a links queue size and requires two parameters, one for the link to make the change to and one for the new queue size. `change_cbr_packetSize` changes the size of packets generated by the cbr specified with the one required parameter. `change_cbr_interval` changes the cbr interval for the cbr specified with the first parameter to the interval specified in the second parameter. Additional detail on each of these commands is contained in Appendix D.

Results of this testing were positive. All the commands outlined above were successfully deployed by the CCT and handled by the enhanced NS-2 software. The commands had near immediate response time, and reflected the offset used with the NetVis instance. Thus, if the offset was set to 1.1 seconds, the effects of sending the command at 5.0 seconds on the visualization clock would take place at 6.1 seconds.

This wired scenario was also successfully tested in a hybrid system experiment. Additional details on this specific environment are explained below.

One additional command was attempted without success. The creation of a new node for the NS-2 simulation was written as a new command and attempted, but had no effect on the simulation. This points out that not all TCL script can be created dynamically, due to the process that NS-2 uses to create the simulation. The best results come from command development evoking dynamic TCL script of scheduled events. If a network event can be scheduled (i.e. `$ns at 6.0 . . . .`) it has a good probability of success in a dynamic TCL environment.

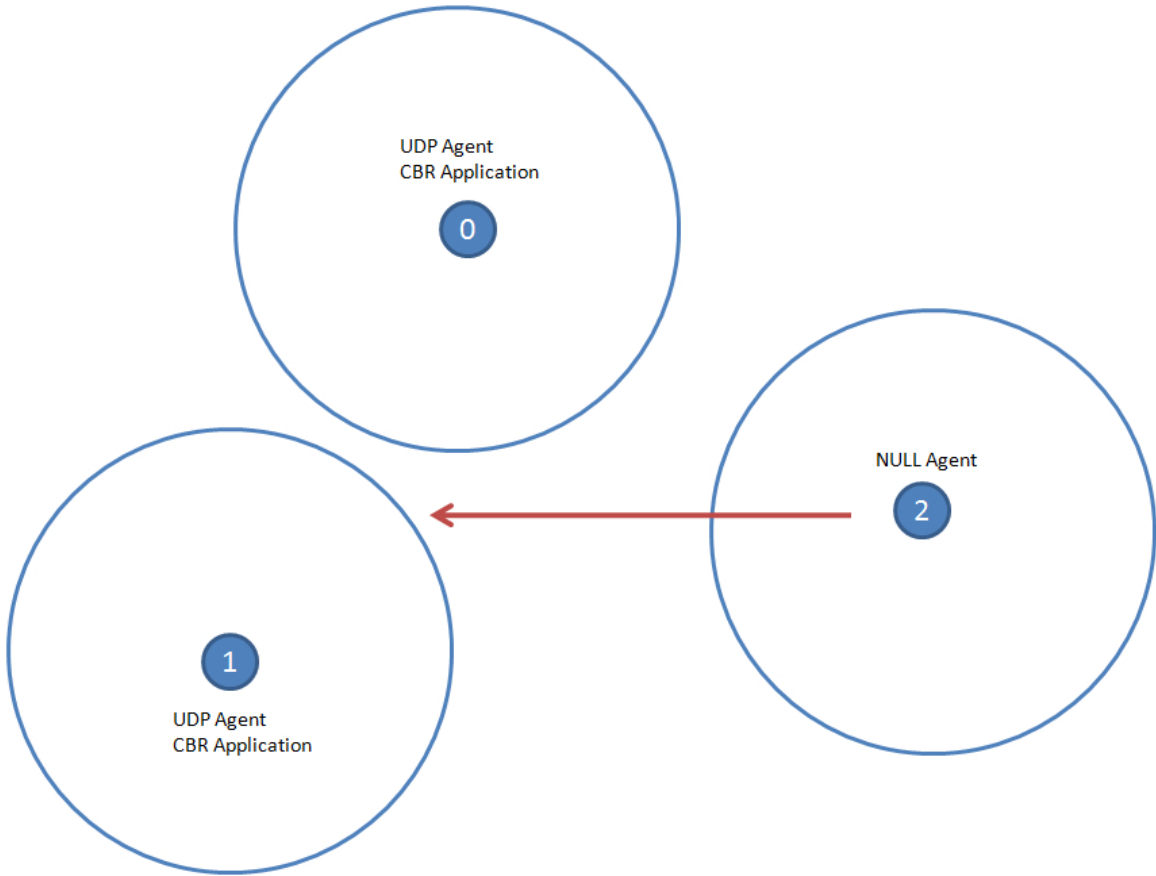


Figure 4.8: Wireless scenario utilized for testing purposes. Three mobile nodes are generated, all with fixed locations. Nodes zero and one attempt to communicate with node two, but all nodes are out of communication range until node two is moved within range of node zero and node one.

*4.2.6.2 Testing Wireless Scenarios.* Wireless scenario commands constructed and testing consisted of tutorial IX.2 *Running Wireless Simulations in ns - Using node movement/traffic-pattern files and other features in wireless simulations* [13]. This simulation consists of three mobile nodes as depicted in Figure 4.8. The node movement was removed from the script to make the scenario generic. This provides a baseline simulation where traffic starts in the beginning and any other actions are the result of command feedback from SimVis or the Configurable Command Tool.

The specific wireless command tested was `move_wireless_node`. `move_wireless_node` moves a node specified and requires two other parameters for x and y coordinates and one for speed. Testing of wireless scenarios was initially attempted with the Clock-Time scheduler, but due to the slow render time of NetVis, the VisSync scheduler was utilized in its stead. This wireless command was also very successful. The command was successfully sent from the CCT and received/handled by the modified NS-2 simulation software. The turn around time from invoking the command to visually displaying the change on NetVis was very similar to the wired commands tested. This particular scenario was run with an offset of five seconds, so the command changes to the simulation were visualized approximately five seconds later. The offset was increased to ensure that the simulation would “synchronize” with NetVis. This offset is not optimal and could probably be reduced without detrimental effects. Additional commands should be developed as a future research effort to validate this finding. Additional details on these commands are in Appendix D.

*4.2.6.3 Testing Combined Air Operations Center Scenarios.* In order to test the system in a comprehensive environment containing a complex scenario, a Combined Air Operations Center (CAOC) scenario was developed. The system composition remained the same with NS-2 as the simulator, the mediator, NetVis as the rendering visualization suite, and the Configurable Command Tool as the command generator. Successful exit criteria for this testing is the successful command

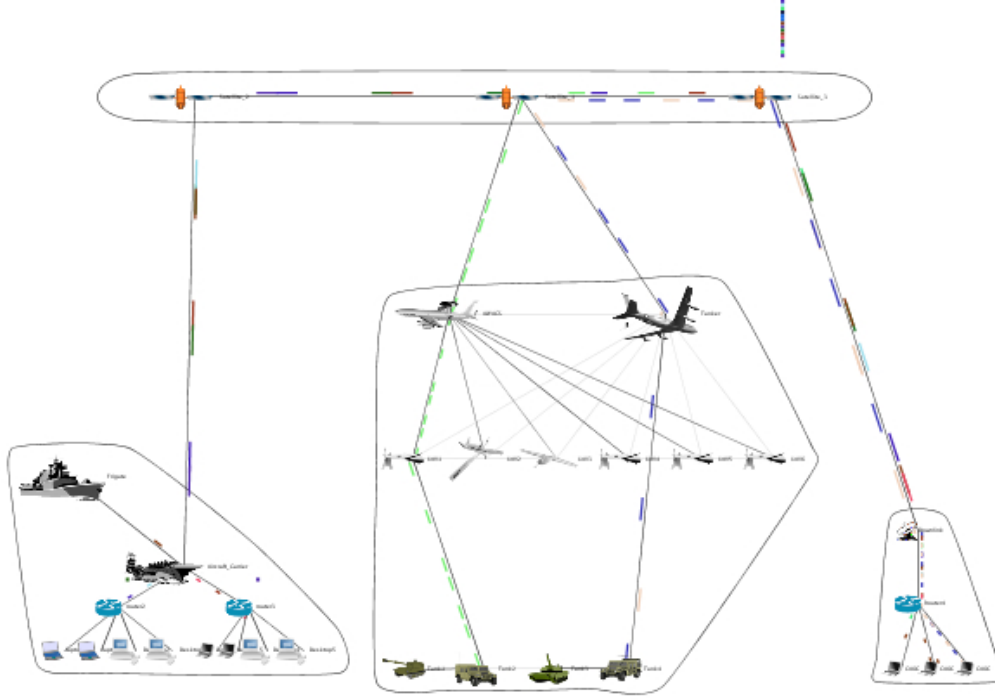


Figure 4.9: CAOC scenario utilized for testing purposes. This scenario consists of non-movable nodes with wired connections, but emulates a realistic network of satellites, aircraft, ships, vehicles, and computers.

feedback generated by the CCT, handled by the modified NS-2, and visually rendered on NetVis. The specific actions are taking a link down and bringing it back up.

Specific scenarios tested consisted of one script. This script utilized only wired non-moving nodes. A depiction of this scenario is in Figure 4.9. This scenario was tested on both a single platform as well as the hybrid system described below. Command feedback tested on this specific scenario was utilizing the `rtmodel-at` syntax to perform the actions of taking a link down and bringing it back up.

This specific testing was not successful for command feedback. The commands were successfully scheduled and handled, but no effect was observed on the simulation. The thought on this failure is that NS-2 pre-calculates the traffic associated with the `rtmodel` and cannot dynamically change the data after the simulation has begun. More investigation into this could be beneficial. This unsuccessful testing refers to the specific `rtmodel` command attempting to take down and bring up a link. The

wired commands developed and tested in Section 4.2.6.1 were not attempted in the CAOC environment, but have the potential for successful use.

The success for testing the CAOC scenarios was found when testing the educational system. The details on this testing are described below.

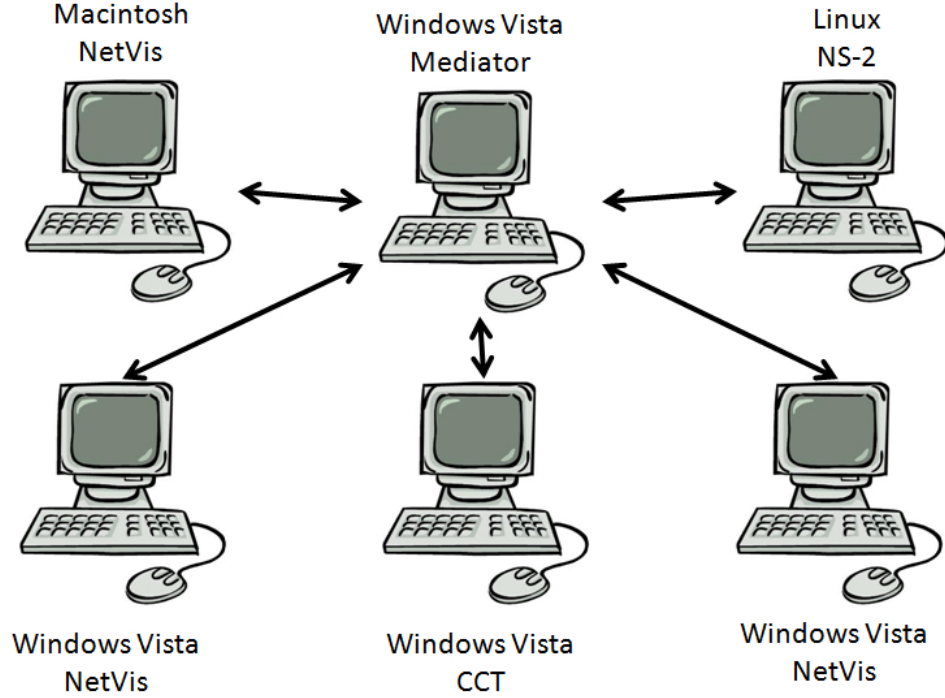


Figure 4.10: Hybrid test system comprised of two Intel Windows Vista laptops, one Macintosh laptop, and one Intel Linux laptop. This test system demonstrates the portability of the simulation system as well as it's capability for remote connection.

*4.2.7 Distributed Hybrid system Testing.* In order to test a system comprised of multiple hardware platforms and operating systems, a hybrid test bed was constructed. The test system built utilized five Intel laptops with Windows Vista, One Intel Laptop running Linux (Ubuntu), and one Macintosh laptop running Mac OS. The specific machines and operating systems is detailed in Section hardware. All machines were connected with an eight port hub and static IP addresses. A depiction of the above system is shown if Figure 4.10.

The goals of this test are to demonstrate that the components on different machines and operating systems can interact the same as a complete system on one machine. The wired scenario and the COAC scenario described above were tested on the hybrid system. Successful exit criteria would be that the system could multiplex data to all visualizations and the CCT could still evoke command feedback. All visualizations should reflect the correct changes to the simulation at the correct time.

The Linux machine hosted NS-2, the Macintosh hosted NetVis, one Windows laptop hosted the mediator, and one Windows laptop hosted the CCT. All software components linked through the common mediator on a Windows laptop.

Two scenarios were tested, first the wired four node scenario described above, then the wired CAOC scenario described above. The four node scenario was successful in displaying a synchronized simulation as well as facilitating interaction through command feedback on the CCT. Each visualization displayed the changes at the correct time. It was noted, however, that the visualizations ran within a few seconds of each other and the fastest one would drive the simulation clock. The wired CAOC scenario successfully displayed a synchronized simulation with the ability to pause and re-start the simulation using NetVis controls. These results are extremely promising and demonstrate the ability to port an interactive simulation system to many platforms including remote locations.

*4.2.8 Educational System Testing.* An educational system was built for testing, comprised of a Linux laptop and five Windows Vista laptops. The Linux laptop hosted the instance of NS-2, CCT, and the mediator. The Windows laptops hosted NetVis. This system was configured to emulate a classroom setting with the instructor on the Linux machine and students on the Windows machines. Each of the students laptops were connected to the instructor's mediator to provide a multiplexed teaching environment. Figure 4.11 illustrates the above system during testing.

Goals for this specific test was to demonstrate that the system could render multiple visualizations all driven and controlled by the instructor's machine. Suc-

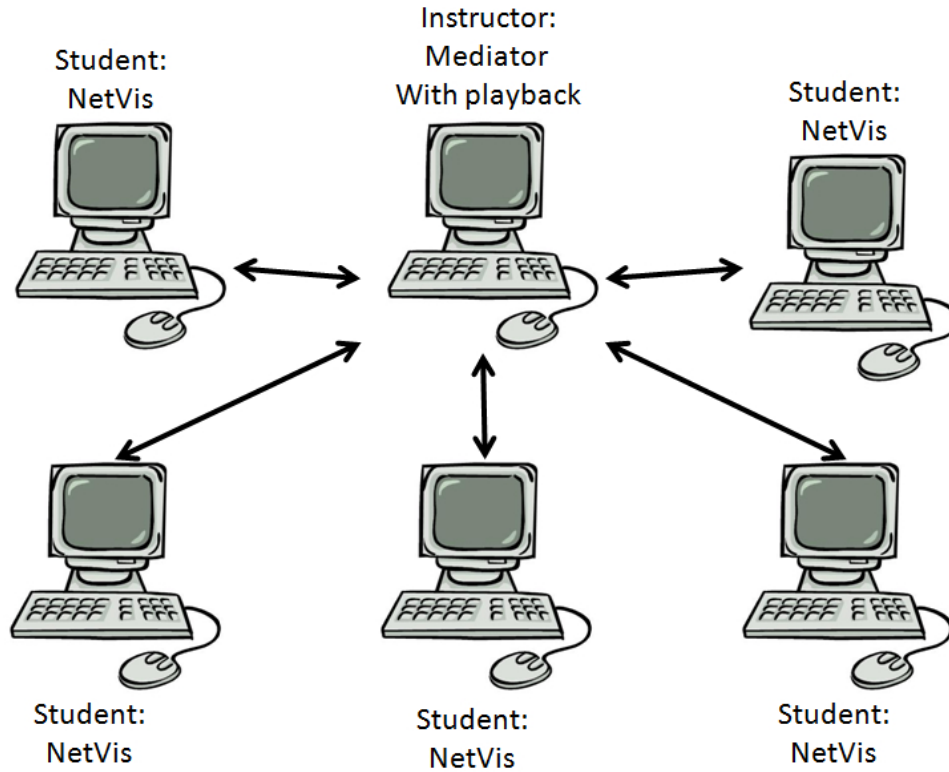


Figure 4.11: Depiction of the educational test system comprised of one Linux laptop and five Windows laptops. The Linux laptop represents the instructor's with NS-2 and the CCT deployed. The Windows laptops represent student laptops with NetVis deployed. The Windows laptops support synchronized display of the simulation.

Successful exit criteria would be the simultaneous display of the same network events on all five student laptops and successful control (i.e. pausing and command feedback) from the instructor's laptop reflected on each of the student laptops. An additional test was also performed to verify the playback capability of the mediator *without* a active simulation. Successful exit criteria for the additional test is the multiplexing of simulation data displayed concurrently on all student laptops.

The four node wired simulation described above was tested on the educational system, with command feedback provided on the instructor's laptop. Each of the student's laptops displayed the same visualization, staying synchronized with the simulation execution. No lag was observed. Command feedback was successful utiliz-

ing the CCT, and simulation changes were displayed simultaneously on all the student laptops.

A similar test was run using the CAOC scenario without command feedback to test the ability for a seminar scenario with capability to pause the simulation. The same hardware/software configuration was utilized. The CAOC scenario was successfully multiplexed to all student Windows machines and the simulation had the capability to be paused and restarted from the CCT and NetVis. All windows machines stayed synchronized with the simulation execution without problems.

One additional system configuration was tested for educational purposes. the system consisted of the five Windows Vista laptops each running NetVis. One laptop hosted the mediator, used to read from a file containing a previously run simulation for distributed scenario playback. All five laptops received the trace file data without error and were able to render the playback of the simulation by executing file read on the mediator.



## V. Contributions and Future Work

This chapter describes the command feedback system components, the contributions of this research, and future work that should be continued after the conclusion of this research. The results of this research contribute to six separate areas: command feedback, multiplexing and co-simulation, a distributed hybrid system capability, synchronized simulation, simulation recording and playback, and simulation interaction without visualization. Many of these areas apply not only to research, but also to education and training. The recommended follow-on work cited below will bring additional capability and efficiency to the command feedback simulation system.

### 5.1 Command Feedback Network Simulation System Components

The system as a whole is depicted in Figure 5.1. It is composed of a simulator (with potential for co-simulation), the mediator, the configurable command tool, and NetVis [3]. Each component has evolved as the research progressed and details for each are outlined below.

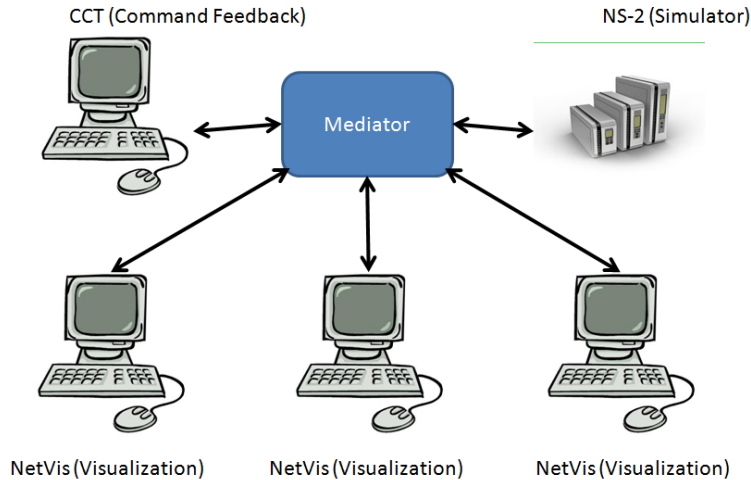


Figure 5.1: Depiction of the command feedback simulation system components. An enhanced NS-2 executes the simulation while the Configurable Command Tool and NetVis interact through the mediator. The system allows multiplexing of communication comprised of simulation data and command feedback.

*5.1.1 Mediator.* The mediator is the heart of the system. Designed as middleware, the mediator couples visualizations and simulators to provide an integrated co-simulation system. As described in Section 3.2.1, the mediator has the capability to record an executing simulation as well as the capability to drive a visualization from the pre-recorded simulation. As shown in Section 4.2.1, the mediator’s ability to multiplex communication, providing the capability to connect multiple simulators and/or visualization suites, was successfully tested.

*5.1.2 NS-2.* NS-2 is an event driven network simulator. This research has enhanced NS-2 code to provide the capability to link with the mediator and stream simulation data to a visualization/s as it is generated. In addition, command reception and handling components have been added to facilitate user interaction with NS-2 as it executes. The execution time of NS-2 can out pace the rendering capabilities of a visualization package and to alleviate this problem, two additional NS-2 schedulers have been written.

*5.1.2.1 Command Handling.* Command reception and handling was developed with a limited number of available commands, but with the potential for a great number of additional commands. The command reception and handling allows a visualization to interact with the simulator as it executes changing desired parameters or causing specific events to occur. This was successfully tested in Section 4.2.4

*5.1.2.2 Schedulers.* Additionally, two new schedulers were developed to enhance the usability of the simulation system. The ClockTime scheduler attempts to throttle NS-2 execution to wall clock time, slowing down the simulation to allow user interaction. The VisSync scheduler was developed for visualization suites that render the simulation events slower than wall clock time by synchronizing the two suites. These new schedulers proved themselves during testing are are detailed in Section 4.2.5.

*5.1.3 Configurable Command Tool.* The CCT started development as the SimVis (see Section 3.4.3), originally developed as intermediate software to aid testing. As testing progressed, the usefulness of the CCT became evident. The CCT is a viable component of the simulation system with configurable command buttons allowing up to six commands to be sent to the simulator to provide user interaction. The CCT is envisioned for use not only by researchers, but also in a classroom setting by the instructor. The CCT was a vital component to testing scenarios as well as the hybrid distributed and educational systems (see Sections 4.2.6, 4.2.7, and 4.2.8).

*5.1.4 NetVis.* NetVis was not directly part of this research, but the parallel efforts of [28] provided opportunity to produce viable results during testing. The additional command feedback capability and TCP/IP connection with the mediator facilitated full system testing of command feedback. Since neither SimVis nor the CCT were capable of rendering events, NetVis was an invaluable resource for testing. The coupling of NetVis also provided hard evidence of the contributions of this research.

## **5.2 Contributions**

The contributions of this research are many. The mediator is the heart of the system and provides many new capabilities such as distributed system capability, multiplexing, and co-simulation. The developed NS-2 code provides interactive simulation by introducing command feedback from a visualization. In addition, a framework for synchronized execution has also been developed by utilizing the additional NS-2 schedulers.

*5.2.1 Command feedback.* Command feedback has been introduced into NS-2, an event driven network simulator. Command feedback provides a user with the ability to issue changes to the simulator from a visualization as the simulation executes. The changes affect the simulation parameters and occur within seconds of

the command issue. Simulation data that follows the command reflect the desired changes providing a new, flexible simulation system.

This new capability provides researchers with a powerful feature to issue commands changing a single simulation that would take multiple simulation runs with traditional simulation. This flexibility is key to saving precious research time and resources. Details on the command feedback design are found in Chapter III.

Another application of command feedback is in the educational field. Command feedback capability allows instructors and students to interact with a network simulation providing a useful training environment. Instructors can inject network events and students can react to the simulation changes. An outline of an educational simulation system was discussed in Section 4.1.4.

*5.2.1.1 Multiplexing and Co-Simulation.* The mediator (see Section 3.1.1) provides the communication backbone of the above simulation system. The mediator provides connections to both simulators (servers) and visualizations (clients). The mediator has the capability to provide multiple connections to each of these types of components, enabling multiplexing and co-simulation.

Multiplexing occurs when at least one simulator is providing data to the mediator and multiple visualizations are connected. When each visualization client connects to the mediator, it adds it to a list. If more than one visualization is connected, it iterates the list and sends a copy of the data to each visualization. This functionality provides the ability to host more than one researcher. This is also very applicable in an educational environment where multiple students can watch or interact with a simulation at the same time. The researchers or students can also be at remote locations, removed from the simulator and/or mediator.

Co-Simulation occurs when more than one simulator is executing in a coupled complex simulation effort. Co-simulation was not attempted directly in this research, however, the capability has been implemented in the design and development of the mediator. The mediator, as in multiplexing above, allows multiple simulators (servers)

to connect. As a simulator connects, it is added to a list. As commands arrive from the visualization/s, they are sent to each simulator in the list providing a multiplexing of commands. This capability could have potential use in research, where large projects require the use of a combined simulation.

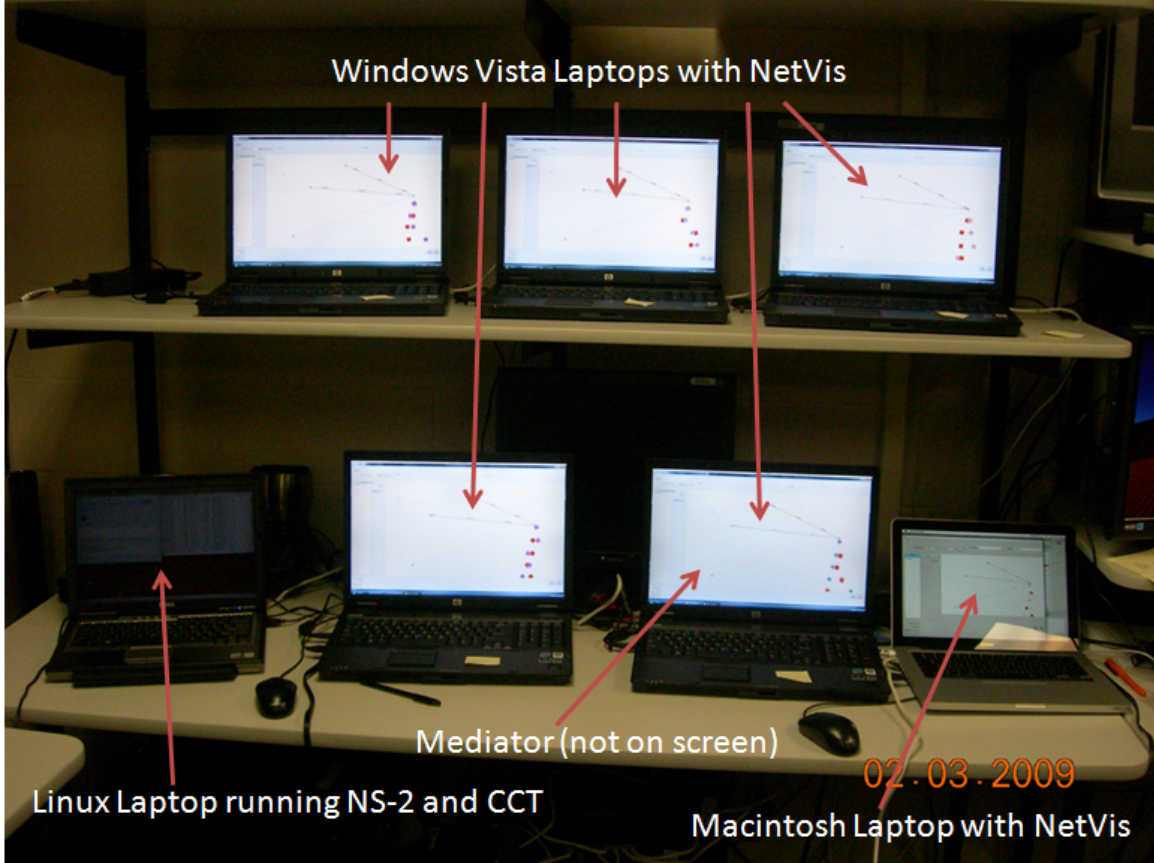


Figure 5.2: Photo of the hybrid test system comprised of five Intel Windows Vista laptops, one Macintosh laptop, and one Intel Linux laptop. This test system employed NetVis, the mediator, and the CCT to achieve command feedback over a distributed hybrid system.

*5.2.2 Distributed Hybrid System Capability.* A distributed system is a collection of interacting components at locations removed from each other. The design of this research effort provides this capability by utilizing TCP/IP connection to connect each component. Both visualizations and simulators connect to a mediator to interact through these TCP/IP connections. This design decision allows the remote connection of visualizations and simulators as long as the IP and Port of the mediator

TCP/IP socket is known. Multiple simulators and visualizations are also supported through remote connections as discussed above.

Not only does TCP/IP allow the use of remote connections, but also provides the capability to span hybrid systems composed of many hardware platforms and operating systems. Not only was TCP/IP utilized for this purpose, but also the decision to use Java as the development language for the mediator and CCT. Java has the capability to be ported to any environment with a Java Virtual Machine capability. As demonstrated in this research (see Section 4.2.7), a hybrid system composed of Intel and Macintosh processor platforms running Linux, Windows, and Mac OS operating systems were supported for a system composed of NS-2, a mediator, a CCT, and six instances of NetVis. Since TCP/IP is a highly supported standard for Internet communication, this research has the potential to be ported to additional platforms. The actual hybrid test system utilized in this research is pictured in Figure 5.2.

*5.2.3 Synchronized System Execution.* Synchronized system execution was an intermediate development of this research. After the initial command feedback capability was implemented in NS-2 and NetVis, it was discovered that NS-2 execution speed far exceeded the rendering capability of NetVis. This caused a situation where the user could not provide the command feedback desired from NetVis before NS-2 finished the simulation. In order to correct this deficiency, two additional schedulers were designed, the ClockTime (see Section 3.3.1) and VisSync (see Section 3.3.2) schedulers.

The ClockTime scheduler attempts to approach wall clock time simulation execution. This initial design was still not sufficient for the coupling of NetVis, since complex simulations took many seconds to render a second of simulation data. To alleviate this, the VisSync scheduler was designed. This scheduler holds NS-2 simulation execution to stay within a specified offset of NetVis' display clock. This provides

the desired capability to interact with the simulation as it executes through command feedback.

These additional schedulers may be useful when integrating other visualizations with the simulation system. The command for setting the offset and clock updates can be utilized by any visualization client to facilitate this process. These specific commands are discussed in detail in Appendix D.

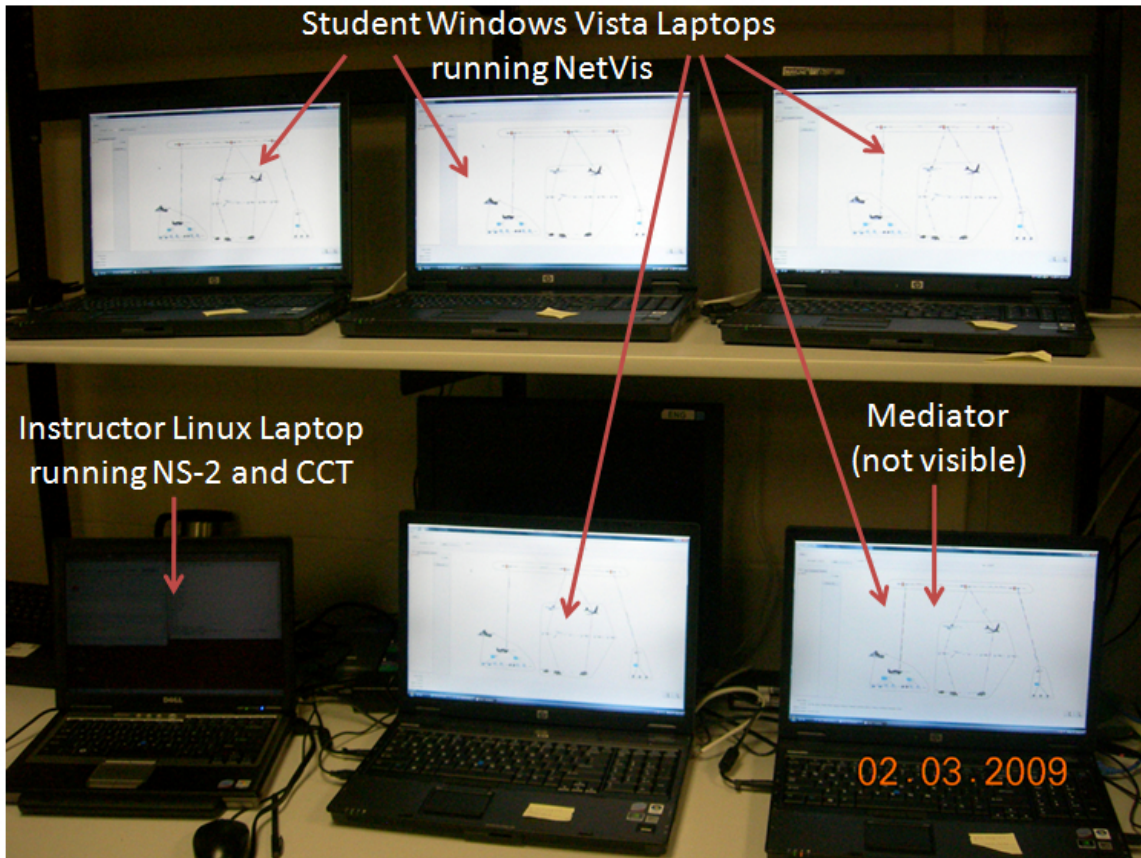


Figure 5.3: Photo of an educational system comprised of five Windows Vista laptops on a network hub displaying a network simulation playback utilizing the mediator’s capability.

*5.2.3.1 Simulation Recording and Playback.* The mediator also provides additional capability to perform simulation recording and playback. This functionality was added to the mediator during the testing phase of this research. The



ability to record a simulation can provide a reference for a researcher or educator to “playback” the simulation at a later time for further analysis.

The playback capability not only allows a previously recorded simulation to be reviewed, but also viewed in multiple visualizations simultaneously using multiplexing (see Section 5.2.1.1). Now, educators can playback a simulation while multiple students view on their screens as the simulation is explained. Remotely located researchers can connect and view simulation results together while on a conference call. This capability was tested and is described in more detail in Section 4.2.8 and a working example system with one instructor laptop and five student laptops is pictured in Figure 5.3.

*5.2.3.2 Simulation Interaction Without Visualization.* The CCT (see Section 4.1.4) provides a capability to interact with an executing simulation without rendering the visualization. Not only can it provide command feedback, but also provides the capability to configure up to six buttons with commands without any change or compilation of code. This tool is applicable in a research environment as well as an educational environment.

The CCT would be useful in a research environment as a command development tool. The researcher could write new command code and test it without deploying a visualization and without modifying the visualization code until the command code is verified. In addition, the visualization and CCT can be deployed in parallel to watch the effects of the command feedback from the CCT on the visualization.

In an educational environment, the CCT can be configured with commands applicable to each lesson plan. Once the CCT is configured, the instructor can invoke the commands applicable to achieve the desired simulation response. Again, as with the research environment described above, the CCT can be deployed in parallel with a visualization allowing the instructor to watch student interactions as well as issue network events utilizing the CCT.



### 5.3 *Future Work*

This section outlines recommended follow-on research work. Due to time limitations of this research, the potential for increased efficiency exists and is outlined in refactoring below. Also suggested are additional command coding and the interfacing of additional simulators and visualization suites to enhance the system's capabilities.

*5.3.1 Refactoring.* This research has the potential for refactoring to improve software design and efficiency of operation. Two areas that would have potential benefit from refactoring are the command string parsing, modifying the CommandParser class to utilize lazy initialization, and command transmission.

*5.3.2 Lazy Initialization.* The CommandParser class contains a singleton instance of each command developed. At the time this iteration of research was completed, this set of commands was small and the code manageable. After additional commands are developed and added to the CommandParser class, memory allocation will increase. There is a solution to this problem - *lazy initialization*.

Lazy initialization, delaying creation objects until they are called, could be a benefit as the number of commands defined increases. As a result, only used commands are instantiated and take up memory resources. So, as the CommandParser receives commands and parameters from the socket, it determines which object is appropriate to schedule as the event handler.

*5.3.3 Command String Parsing.* The Listener class contains the code that parses the command strings sent from visualization suites through the mediator. This parsing code was initially written due to problems with the TCP/IP socket. After refactoring the socket code, the problems requiring this parsing method were no longer encountered. Segmentation faults occasionally occur due to this code when an invalid command is encountered.

*5.3.4 Command Transmission.* The potential exists to improve command transmission. Time should be spent to determine the best method of transmitting short strings over a TCP/IP connection without the use of code such as `println`.

*5.3.5 NetVis Synchronization.* NetVis sends an offset command to ensure NS-2 execution stays slightly ahead of the visualization rendering as clock updates are received. There is a problem with the existing design of NetVis clock updates. Currently, the clock updates only occur when there is network events that require NetVis to continue execution. If there is a “blackout” period anywhere in the simulation without network events for a time period greater than the offset value, deadlock will occur. Deadlock exists due to the lack of clock updates from NetVis which causes NS-2 to wait for NetVis to catch up. Since the simulation pauses and no data is passed, NetVis waits on NS-2 -thus a deadlock state.

To alleviate this problem, a different method of sending clock updates must be developed. Possibly a timeout that checks if the pause button is not depressed, then advances the clock. This problem deserves additional thought.

*5.3.6 Additional Development.* Additional development and integration of components of this system will greatly enhance the usefulness of this research. Three primary areas of additions are recommended for future work: command development, visualization suite integration, and additional simulator integration.

*5.3.6.1 Commands.* Additional command development is highly encouraged for future work. Each additional command developed adds new capability to the existing system. This research only skimmed the surface of wired and wireless commands. To promote interactive research simulation, additional command capability is required. The development process for commands is outlined in Appendix E. This section outlines all pertinent software modules and classes and provides examples of command code.

*5.3.6.2 Visualization and Simulator Suite Integration.* Another area of importance is the interfacing of visualization and simulation suites. The mediator was designed as generic middleware, allowing the interaction of any software that follows the interface document. This provides the potential for additional simulation and visualization capability as well as hybrid systems with multiple simulators and or visualizations. The interface document for the mediator is found in Appendix C. This section outlines the protocol strings and methods required to add a simulator or visualization to the system. As with command development, additional simulators and/or visualization suites will improve the capability of the overall system.

## *Appendix A. System Requirements and Startup Procedures*

This appendix details the procedures required for successful operation of a command feedback enabled system. The system described is comprised of a mediator and one or more of the following components: NS-2 simulator, NetVis, and CCT. The CCT is not necessary, but optional for a complete simulation system.

### ***A.1 System Requirements***

This section details the hardware and operating systems utilized successfully during this research. The application is farther reaching than what we tested, for Java is a “universal” code able to be used on anything that has a Java Virtual Machine installed. The mediator, CCT, and NetVis are written in Java. The simulator, NS-2, is written in C++ and uses a TCL script to build scenarios. Due to this language choice, the code must be compiled on the specific machine type where it will be deployed. Specific details about compiling NS-2 with command feedback is detailed in Appendix B.

*A.1.1 Hardware.* The system was successfully run on the following platforms: AMD Athlon 64 X 2 Dual Core Processor 5000+ personal computer, Intel Core 2 Duo personal computer, and MacBook Intel Core 2 Duo. The system was also run as a distributed system comprised of Intel and Macintosh hardware platforms linked through the TCP/IP sockets of the mediator. All components were successfully run on each set of hardware.

*A.1.2 Operating Systems.* The operating systems that successfully compiled and executed the command feedback system include the following systems: Windows *Vista* [14], Macintosh OS *Leopard* [2], and Linux - Ubuntu *Hardy Heron* [19]. Each operating system was tested to support the entire system and also as a distributed system. The one exception was the Windows Vista operating system did not directly support the compilation of NS-2. Cygwin [21] could be used in this environment, but testing with Cygwin was not performed during this research.

*A.1.2.1 Linux.* Ubuntu was utilized for the development of the NS-2 code extensions to support command feedback. It also supported a full system as well as a portion of the distributed system. Specific Ubuntu builds used are: 8.04 *Hardy Heron*, 8.02 *Hardy Heron*, and 8.10 *Intrepid IbeX*.

*A.1.2.2 Windows.* Windows Vista was utilized as a portion of the distributed system to support the mediator, CCT, and NetVis. Cygwin was not installed to attempt NS-2 execution, although it would be a possibility for full system operation in a Windows environment. Specific builds used were: Windows Vista Home Premium service pack 1 and Windows Vista Business service pack 1.

*A.1.2.3 Macintosh.* The Macintosh operating system successfully used in this research was Macintosh OS 10.5 *Leopard*. This configuration supported full system operation as well as a portion of the distributed system.

*A.1.3 Java Virtual Machine.* The Java [23] version required for successful execution of the mediator, CCT, and NetVis is JRE System Library 1.6.0 or greater. If this version is not installed, the above applications are not expected to run properly if at all.

*A.1.4 Integrated development environment.* One integrated development environment was utilized for development of the mediator and CCT, Eclipse [18]. Eclipse was used to develop NetVis enhancements as well, and is required for the execution of NetVis. The mediator and CCT have the capability to be jarred and run as an executable as well as evoking from Eclipse.

## ***A.2 IP configuration***

IP is configured on four modules. The mediator should be run locally on the machine intended for its use. It opens a port on the host machine with a port of 1234. To change the port number, the variable is *ssocket* in the class Controller. The

CCT asks the user if IP *localhost* and port *1234* are correct, if no is selected, IP and port are entered in dialog windows. NS-2 has two TCP/IP connections. The first is defined in the TCL script (see Figure 4.1). The other port is defined in the *Listener* class of the C++ code and requires recompilation. The last IP to change is contained in NetVis. NetVis has a configuration text file with the IP, but the port is contained in the code.

### ***A.3 System Startup and Operation***

The order of system component startup is essential for a successful simulation. Sequence of application execution to ensure success is:

- Start the mediator.
- Start the CCT (optional) and connect to the mediator.
- Start NetVis and select streaming for as many instances as desired.
- Start NS-2 (the simulation will pause at the initial offset when using the VisSync scheduler).
- Push the “start” button on all NetVis instances to start the simulation.

## *Appendix B. NS-2.32 Code Changes and Building Instructions*

The following changes are made to ns allinone package version 2.32 under the directory *ns-allinone-2.32\ ns-2.32\*. In addition to this package, the Boost::Threading library is also required and should be installed before compiling NS-2.

### ***B.1 Changes to Makefile.in***

Add the following line as the second to last line below *OBJ\_CC = \*:

- `command_controller/CommandHandlers.o command_controller  
/CommandParser.o command_controller/Listener.o \`

Compile the makefile by typing “*configure*”).

### ***B.2 Changes to Makefile***

Add the following text to the line under *LIB = \* :

Note: this line may differ based on the specific operating system and boost library installed.

Note 2: This has to be accomplished after any configure of Makefile.in is accomplished.

- `-lboost_thread-gcc42-mt-1_36.`

### ***B.3 Addition of Command Controller Package***

Add the folder *command\_controller* with the following files:

- `CommandHandlers.h, CommandHandlers.cc, CommandParser.h,  
CommandParser.cc, Listener.h, and Listener.cc.`

#### ***B.4 Changes to the Common Folder***

Make the following changes under the common folder:

- Replace scheduler.h and scheduler.cc.

#### ***B.5 Changes to the tcl\lib Folder***

Make the following changes under the tcl\lib folder:

- Replace ns-lib.tcl file.
- Add ns-listener.tcl.

#### ***B.6 Boost Library Threading***

The threading portion of the Boost Library [4] must be installed on the system and the library added to the PATH.



## *Appendix C. Interface Document*

This appendix outlines the interface requirements for simulation and visualization coupling with the mediator. This information is presented as a guide only and specific software packages interfaced with the mediator will require additional code and or tailoring of the presented material.

### *C.1 Sockets*

Three instances of TCP/IP sockets are used by the mediator, two for a simulation and one for a visualization. The two utilized by the server consist of one for data transmission and one for command reception. In this research, NS-2 creates the data socket in the TCL code and the command reception socket in C++ (see Figure 3.5).

The socket utilized by visualizations has a dual purpose, data reception and command transmission. Both the CCT and NetVis sockets are written in Java.

### *C.2 Protocol*

The protocol used to communicate with the mediator is very limited. The only requirement for protocol strings are at connection and disconnection of the TCP/IP sockets. The four specific protocol strings are listed below.

*C.2.1 Connection Protocol.* Connection protocol strings are used to tell the mediator what type of application is connecting and what purpose the socket will fulfill. Three different protocol strings are used for connection initiation.

*C.2.1.1 server.* The connection protocol string ‘**server**’ tells the mediator that a simulation is connecting. The specific use of this socket is data transmission from the simulator to the clients. No data can be transmitted until the string ‘**server**’ is sent by the simulation.

*C.2.1.2 client.* The connection protocol string ‘**client**’ tells the mediator that a visualization is connecting. The specific use of this socket is data

reception from the simulator as well as command transmission to the simulators. No data can be received nor commands sent until the string ‘‘client’’ is sent by the visualization instance.

*C.2.1.3 command.* The connection protocol string ‘‘command’’ tells the mediator that a simulation is connecting. The specific use of this socket is command reception from the visualizations. No commands can be received until the string ‘‘server’’ is sent by the simulation.

*C.2.2 Disconnection Protocol.* There is only one protocol string utilized for disconnection, ‘‘Goodbye’’. This protocol string should be sent before any simulator or visualization disconnects its TCP/IP connection. This protocol is repeated to the appropriate visualization/s or simulator/s to announce that there was a disconnection.

### ***C.3 Synchronization***

Synchronization is a delicate problem and required specific tweaking of the offset parameter to optimize simulation execution with simultaneous visualization rendering. It relates the visualization and simulator. The following changes were made in NS-2.

*C.3.1 ClockTimeScheduler.* The ClockTime Scheduler runs independent of the visualization and attempts to parallel wall clock time. Due to the independent nature of this scheduler, no specific synchronization is required for use. To invoke this scheduler, see Section 3.3.1

*C.3.2 VisSyncScheduler.* The VisSyncScheduler requires constant ‘‘heart-beats’’ from a visualization to tell the simulation the time it can advance to before waiting for another update. To invoke this scheduler, see Section 3.3.2. The specific command is outlined in Appendix D.3.3. An offset was utilized in conjunction with the clock pulses to ensure that NS-2 stayed ahead of NetVis. The offset command

is described in Appendix D.3.4. The offset command is sent by NetVis when the start button is depressed. The offset is tailored for each individual scenario. Use of this scheduler in our specific research required a limitation to the amount of updates issued by NetVis. The default optimized parameters for a simple wired scenario was sending a clock update every second with an offset of 1.1 seconds. The offset is tied to the amount of “blackout” without network traffic. NetVis only sends clock updates while rendering network events, so if there is a time with no events, the clock updates are not sent. If the offset is greater than this “blackout” then the simulation will continue, otherwise deadlock will occur.

#### ***C.4 Commands***

Commands are passed along with their associated parameters as strings separated by spaces and handled by the parser in Listener.cc. The mediator acts as a repeater/multiplexer and passes the commands on to the simulator.

## Appendix D. Commands Developed

This appendix lists the commands written for NS-2 command handling. The commands include both wired and wireless specific items.

### D.1 Wired Commands

Wired commands were specifically written for wired scenarios, but can be utilized in a hybrid wired/wireless scenario to affect the wired portion of the simulation. Six commands have been written for the wired portion of this research: `change_queue_size`, `turn_off_cbr`, `turn_on_cbr`, `change_cbr_packetSize`, `change_cbr_interval`, and `move_cbr`.

*D.1.1 change\_queue\_size.* The `change_queue_size` command is used to change the buffer size of a link. It's TCL equivalent is `queue-limit` and it requires three parameters, two for the nodes at each end of the link and one for the queue-limit size. Example :

```
change_queue_size n1 n2 25 0 0
```

is the command to change the buffer size of the link between nodes one and two to 25. The dynamic TCL generated would be `$ns at (current time) "$ns queue-limit $n1 $n2 25"`.

*D.1.2 turn\_off\_cbr.* The `turn_off_cbr` command stops the transmission of a cbr. It's TCL equivalent is `stop` and it requires one parameter for the specified cbr. Example:

```
turn_off_cbr cbr1 0 0 0 0
```

stops the transmission of packets at cbr1. The dynamic TCL generated would be `$ns at (current time) "$cbr1 stop"`.

*D.1.3 turn\_on\_cbr.* The `turn_on_cbr` command starts the transmission of a cbr. It's TCL equivalent is `start` and it requires one parameter for the specified cbr. Example:

```
turn_on_cbr cbr1 0 0 0 0
```

starts the transmission of packets at cbr1. The dynamic TCL generated would be `$ns at (current time) '$cbr1 start'` .

*D.1.4 change\_cbr\_packetSize.* The `change_cbr_packetSize` command changes the size of the packets transmitted by a specific cbr. It's TCL equivalent is `set packetSize_` and it requires two parameters, one for the cbr of interest and one for the new packet size. Example:

```
change_cbr_packetSize cbr1 500 0 0 0
```

changes the packet size of cbr1 to 500. The dynamic TCL generated would be `$ns at (current time) '$cbr1 set packetSize_ 500'` .

*D.1.5 change\_cbr\_interval.* The `change_cbr_interval` command changes the interval between packet transmissions at the specified cbr. It's TCL equivalent is `set interval_` and it requires two parameters, one for the cbr of interest and one for the new interval time. Example:

```
change_cbr_interval cbr1 0.25 0 0 0
```

changes the transmission interval time of cbr1 to 0.25 seconds. The dynamic TCL generated would be `$ns at (current time) '$cbr1 set interval_ 0.25'` .

*D.1.6 move\_cbr.* The `move_cbr` command attaches a cbr to an agent in the simulation. This in effect moves the cbr from it's current location to a new location in the topology. It's TCL equivalent is `attach-agent` and it requires two parameters, one for the cbr of interest and one for the agent to attach the cbr to. Example:

```
move_cbr cbr1 udp2 0 0 0
```

moves cbr1 from its current agent to udp2. The dynamic TCL generated would be `$ns at (current time) '$cbr1 attach-agent $udp2'` .

## D.2 Wireless Commands

Wireless commands were specifically written for wireless scenarios, but can be utilized in a hybrid wired/wireless scenario to affect the wireless portion of the simulation. One command has been written for the wireless portion of this research: `move_wireless_node`.

*D.2.1 move\_wireless\_node.* The `move_wireless_node` command moves a wireless node to a new grid location at a specified speed. Its TCL equivalent is `setdest` and requires four parameters, one for the node of interest, one for the x coordinate, one for the y coordinate, and one for the speed. Example:

```
move_wireless_node n1 100 150 25 0
```

moves node n1 to 100, 150 at speed 25. The dynamic TCL generated would be `$ns at (current time) "$n1 setdest 100 125 25"` .

## D.3 Non-specific Commands

Non-specific commands were developed for for control of the scheduler. These special case commands do not require a handler and thus are not scheduled. The method is invoked in *CommandParser.cc* which has a handle to the scheduler instance. Four commands have been written for the scheduling control portion of this research: `pause`, `resume`, `update_vis_clock`, and `set_read_ahead_offset` .

*D.3.1 pause.* The `pause` command pauses the scheduler execution until a resume is issued. This command requires no parameters. No events will be dispatched while the scheduler is paused. Example:

```
pause 0 0 0 0 0
```

pauses the scheduler. The specific scheduler command is `Scheduler::instance().pause()` .

*D.3.2 resume.* The `resume` command resumes the scheduler execution after a pause is issued. This command requires no parameters. This continues the

dispatch of events where the scheduler left off when paused. Example:

```
resume 0 0 0 0 0
```

resumes scheduler execution. The specific scheduler command is

```
Scheduler::instance().resume() .
```

*D.3.3 update\_vis\_clock.* The `update_vis_clock` command updates the `visClock_` variable in *scheduler.h* to the current visualization time and requires one parameter for the time. This command is used in conjunction with `set_read_ahead_offset` to throttle the scheduler execution. Example:

```
update_vis_clock 24.765 0 0 0 0
```

updates the `visClock_` variable to 24.765 (seconds). The specific scheduler command is `Scheduler::instance().updateVisClock(newTime)` .

*D.3.4 set\_read\_ahead\_offset.* The `set_read_ahead_offset` command updates the `readAheadOffset_` variable in *scheduler.h* to a time the visualization requests. This command requires one parameter for the time. This command is used in conjunction with `update_vis_clock` to throttle the scheduler execution and ensures the simulation stays ahead of the visualization. Example:

```
set_read_ahead_offset 1.0 0 0 0 0
```

updates the `readAheadOffset_` variable to 1.0 (seconds). The specific scheduler command is `Scheduler::instance().setReadAheadOffset(offset)` .

## Appendix E. Command Development

This appendix outlines the command development and test process. Command development begins with modification of two C++ (.cc) files and two header files (.h). After command code is written, two stages of testing are recommended, script testing and testing with the Configurable Command Tool.

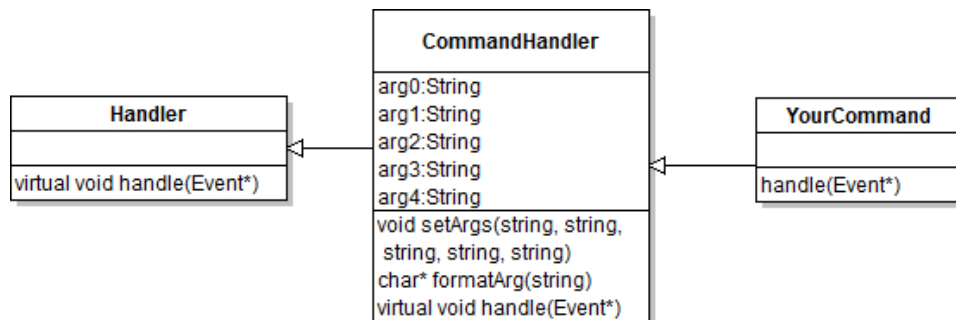


Figure E.1: Illustration of the inheritance tree of a command. The CommandHandler class contains the overhead code to simplify the design of new commands by only requiring the development of the handle() method.

### E.1 Developing commands

The .cc and .h files that require modification for command development are located in the nsallnone/command\_controller folder. The CommandParser modifications refer to *CommandParser.h* and *CommandParser.cc*. The CommandHandler modifications refer to *CommandHandler.h* and *CommandHandler.cc*.

Each command inherits from the `CommandHandler` class, which inherits from the `Handler` class as depicted in Figure E.1. The CommandHandler class takes care of the overhead and allows the development of the command specific `handle()` event.

Command development, however, does not begin with writing C++ code. It begins by writing TCL script. Every command is basically a dynamic TCL script created and passed to the TCL instance for immediate execution. If an event can be accomplished by writing it in a TCL script, then there is an excellent chance that it can be created dynamically through command handling. Not all TCL can



be translated to a dynamic command. Each command will have to be individually written and tested to verify it's correctness for use in the command feedback system.

```

else if (command.compare("change_queue_size") == 0){
    if (debug > 0) {
        cout << "change_queue_size received with parameters
            \",\",
            << a0 << "\",\" \",\",
            << a1 << "\",\" \",\",
            << a2 << "\",\" \",\",
            << a3 << "\",\" \",\",
            << a4 << "\",\" at time \" << current_time
            << "\\n\";
    }
    queueSizeHandler.setArgs(a0, a1, a2, a3, a4);
    Scheduler::instance().schedule(&queueSizeHandler,
        new Event(), offset);
}

```

Figure E.2: Example code for command development.

*E.1.1 CommandParser code.* *CommandParser.h* contains the singleton instances of all command handlers. Any new command developed that has a `handle()` method must have an instance declared in this file. Naming conventions are describe in Section E.2.

*CommandParser.cc* contains the code to determine which command was sent from a client. They are all contained within a block of if/then/else statements. A new else if block will contain your code. There is are four lines to write and an example is depicted in Figure E.2. The first compares the passed string with your command in quotes. The second line is a debug statement that lets the user know which command was received. The third line sets the arguments passed in with the command to the appropriate handler you defined in *ComandParser.h*. The fourth line schedules the handlers that will be called to process your command. The third and fourth lines may need to be replicated if you need to call more than one command handler as in `change_cbr_packetSize` in Figure E.3.

```

else if (command.compare("change_queue_size") == 0){
    if (debug > 0) {
        cout << "change_queue_size received with parameters
            \\", \"\",
            << a0 << \"\", \"\",
            << a1 << \"\", \"\",
            << a2 << \"\", \"\",
            << a3 << \"\", \"\",
            << a4 << \"\" at time \"\",
            << current_time << \"\\n\";
    }
    turnOffCBRHandler.setArgs(a0, a1, a2, a3, a4);
    cbrPacketSizeHandler.setArgs(a0, a1, a2, a3, a4);
    turnOnCBRHandler.setArgs(a0, a1, a2, a3, a4);
    Scheduler::instance().schedule
        (&turnOffCBRHandler, new Event(), offset);
    Scheduler::instance().schedule
        (&cbrPacketSizeHandler, new Event(), offset);
    Scheduler::instance().schedule
        (&turnOnCBRHandler, new Event(), offset);
}

```

Figure E.3: Example command code requiring multiple setArgs() and scheduled handlers. Before the packet size of a cbr is changed it must be turned off. After the change is made, the cbr can be turned on again. Three events are scheduled, each with their own handler.

*E.1.2 CommandHandler code.* *CommandHandlers.h* contains the declaration of each command handler class. Each class contains only one method, `handle(Event*)`. Each command handler inherits from the `CommandHandler` abstract class.

*CommandHandlers.cc* defines the `setArgs()` and `formatArg()` methods which is inherited from the `CommandHandler` class. It is also the file that defines the specific `handle()` methods for all command handlers. Figure E.4 outlines an example command for turning off a cbr. The important line in the handle method is the `sprintf` statement. This line defines the dynamic TCL that will be executed to handle the developed command. Not all TCL events can be created dynamically.

```

void TurnOffCBRCommand::handle(Event* event){
int debug = 1;
if (debug > 0){
    cout << "Handling TurnOffCBRCommand at time "
        << Scheduler::instance().clock() << "\n";
    cout << "setArgs : Arg 0 is " << arg0 << "Arg 1 is "
        << arg1 << " Arg 2 is " << arg2 << " Arg 3 is "
        << arg3 << " Arg 4 is " << arg4 << "\n";
}
Tcl& tcl = Tcl::instance();
char parameters[128];
sprintf(parameters, "$ns at %f \"'$s
    stop\''\", Scheduler::instance().clock(), formatArg(arg0));
if (debug > 0){
    cout << "TurnOffCBRCommand parameters are : "
        << parameters << "\'\'n";
}
tcl.eval(parameters);

```

Figure E.4: Example handler code illustrating the sprintf statement that creates the dynamic TCL passed to the TCL instance for a turn off cbr command. The dynamic TCL line is passed to the TCL instance in the last line of the code.

One known shortfall is the creation of new nodes cannot be accomplished dynamically. Commands will require individual testing to see if they are viable in a dynamic TCL script.

## E.2 Naming conventions

Specific naming conventions were used for certain items in the NS-2 C++ code. The first is the singleton instances of command handlers contained in *CommandParser.h* and *CommandParser.cc*. The instances of each command handler are named to reflect the command and add the word *Handler* on the end. For example, the turn off cbr command handler singleton instance is named *turnOffCBRHandler*.

The other application of a specific naming convention is the class names for each of the command handlers located in *CommandParser.h* and *CommandHandlers.cc*.

The classes are again named after the command that they handle plus the word *Command* added on the end. For example, the turn off cbr command handler class is named *TurnOffCBRCommand*.

### ***E.3 Testing commands***

Testing commands is an important part of command development. The testing is accomplished in phases as the command code is developed starting with TCL script testing and progressing to full system testing.

*E.3.1 TCL script testing.* TCL script testing is accomplished in two separate steps. One is part of the initial command development by writing a TCL script to accomplish the desired response. If the action cannot be accomplished in a general TCL script, it cannot be developed as a viable command. As stated above, not all TCL script will work correctly as in a dynamic environment, but there is a good probability that it will.

Once the TCL script is providing the desired response the command code is developed. The command code can be tested solely using NS-2 instead of a complete system. This is accomplished by instantiating a **Listener** object in the tcl script, then evoking the command through this Listener object. An example of this scripting action is outlined in Section 3.2.2.2.

```
turn off cbr1
turn_off_cbr cbr1 0 0 0 0
```

Figure E.5: Example CCT command.cfg code to create a turn off cbr button for testing. The first line creates the button text and the second line defines the command string sent to NS-2 for execution. This specific command turns off cbr1 when the button is pressed. The zeros are padding since the required amount of parameters is five.

*E.3.2 CCT testing.* Once the command code is developed and tested within a TCL script, the command is ready for full system testing using the CCT. The CCT's

command.cfg file is changed to reflect the new command/s available for testing. Two lines are used for each command. The first line defines the button text on the CCT's GUI. The second line should be the actual string that will be sent to NS-2 for command handling including the command, parameters, and padding if necessary. An example of defining a button for turning off a cbr is shown in Figure E.5.

*E.3.3 NetVis testing.* NetVis testing is the last stage of the testing process. This would also be accomplished if another visualization was used in the place of NetVis. The visualization software will require changes that are not covered specifically in this research. Once the changes are completed, the system can be tested by invoking the command from the visualization rather than the CCT. This is the desired end product of command development and is highly encouraged.

## Bibliography

1. Andrs Varga. “Network Animator (NAM)”, Page last accessed on Nov 4, 2008. [Http://www.omnetpp.org/](http://www.omnetpp.org/).
2. Apple Inc. “Mac OS X Leopard”, Page last accessed Nov 4, 2008. [Www.apple.com/macosx/](http://www.apple.com/macosx/).
3. Belue, J. Mark. *Network Visualization Design Using Prefuse Visualization Toolkit*. Master’s thesis, AFIT, March 2008.
4. Boost.Org. “Boost C++ Libraries”, Page last accessed on Nov 4, 2008. [Http://www.boost.org/](http://www.boost.org/).
5. Branicky, Michael, Vincenzo Liberatore, and Stephen M. Phillips. “Networked Control System Co-Simulation for Co-Design”. *Proceedings of the American Control Conference*, 143–150. ACM, New York, NY, USA, 2003. ISBN 1-59593-188-0.
6. Fujimoto, Richard M. *Parallel and Distributed Simulation Systems*. Wiley, 2000.
7. Group, VINT Research. “The NS Manual (formerly NS Notes and Documentation)”, Page last accessed on Nov 9, 2008. [Http://www.isi.edu/nsnam/ns/doc/node626.html](http://www.isi.edu/nsnam/ns/doc/node626.html).
8. Henderson, Thomas R. and Sumit Roy. “ns-3 Project Goals”. *Fix this cite*, 205–217. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24473-8.
9. Hopkinson, Kenneth, Xiaoru Wang, Renan Giovanini, James Thorp, Kenneth Birman, and Denis Coury. “EPOCHS: A Platform for Agent-Based Electric Power and Communication Simulation Built From Commercial Off-the-Shelf Components.” *IEEE Transactions on Power Systems*, 21(2):p548 – 558, 20060501. ISSN 08858950.
10. Kurkowski, Stuart, Tracy Camp, and Michael Colagrosso. “A Visualization and Animation Tool for NS-2 Wireless Simulations: iNSpect”. *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 503–506, 2005.
11. LinuxHowtos.org. “Socket Tutorial”, Page last accessed on Nov 9, 2008. [Http://www.linuxhowtos.org/C\\_C++/socket.htm](http://www.linuxhowtos.org/C_C++/socket.htm).
12. Mahrenholz, Daniel and Svilen Ivanov. “Adjusting the ns-2 Emulation Mode to a Live Network”. *Proceedings of Communication in Distributed Systems 2005*, 205–217. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24473-8.
13. Marc Greis. “Tutorial for the Network Simulator “ns””, Page last accessed Nov 4, 2008. [Www.isi.edu/nsnam/ns/tutorial/](http://www.isi.edu/nsnam/ns/tutorial/).
14. Microsoft. “Windows Vista”, Page last accessed Nov 4, 2008. [Http://www.microsoft.com/windows/windows-vista/default.aspx](http://www.microsoft.com/windows/windows-vista/default.aspx).

15. Microsystems, Sun. “Writing the Server Side of a Socket”, Page last accessed on Nov 9, 2008. [Http://java.sun.com/docs/books/tutorial/networking/sockets/clientServer.html](http://java.sun.com/docs/books/tutorial/networking/sockets/clientServer.html).
16. Multiple Authors. “NS-2 Contributed Code”, Page last accessed on Nov 4, 2008. [Http://nsnam.isi.edu/nsnam/index.php/Contributed\\_Code](http://nsnam.isi.edu/nsnam/index.php/Contributed_Code).
17. Myles, Ashish. “Java TCP Sockets and Swing Tutorial”, Page last accessed on Nov 9, 2008. [Http://www.ashishmyles.com/tutorials/tcpchat/](http://www.ashishmyles.com/tutorials/tcpchat/).
18. Open Source. “Eclipse”, Page last accessed Nov 4, 2008. [Www.eclipse.org/](http://www.eclipse.org/).
19. Open Source. “Ubuntu”, Page last accessed Nov 4, 2008. [Http://www.ubuntu.com/](http://www.ubuntu.com/).
20. Opnet Tech. “OPNET”, Last accessed on Nov 4, 2008. [Www.OPNET.com/](http://www.OPNET.com/).
21. Red Hat. “Cygwin”, Page last accessed Nov 4, 2008. [Http://www.cygwin.com](http://www.cygwin.com).
22. Riley, George F. “The Georgia Tech Network Simulator”. *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, 5–12. ACM, New York, NY, USA, 2003. ISBN 1-58113-748-8.
23. Sun Microsystems. “Java”, Page last accessed Nov 4, 2008. [Http://java.com/en/](http://java.com/en/).
24. University of California, Los Angeles. “Global Mobile Information Systems Simulation Library”, Last accessed on Nov 4, 2008. [Http://pcl.cs.ucla.edu/projects/glomosim/](http://pcl.cs.ucla.edu/projects/glomosim/).
25. VINT Research Group. “The Network Simulator (NS-2)”, Page last accessed on Nov 4, 2008. [Http://nsnam.isi.edu/nsnam/](http://nsnam.isi.edu/nsnam/).
26. VINT Research Group. “OMNeT++”, Page last accessed on Nov 4, 2008. [Http://www.isi.edu/nsnam/nam/](http://www.isi.edu/nsnam/nam/).
27. Warren, Gary, Ronald Nolte, Ken Funk, and Brian Merrell. “Network simulation enhancing network management in real-time”. *ACM Trans. Model. Comput. Simul.*, 14(2):196–210, 2004. ISSN 1049-3301.
28. Yetisti, Cigdem. *What is the title*. Master’s thesis, AFIT, March 2009.

## *Index*

The index is conceptual and does not designate every occurrence of a keyword. Page numbers in bold represent concept definition or introduction.

- cbr, 8
- CCT, 38
- ClockTime scheduler, 28, 45
- Co-Simulation, 9
- CommandHandler, 27
- CommandParser, 27
- Command Configuration Tool, *see* CCT
- Communication Mediator, *see* mediator
- Constant Bit Rate, *see* cbr
- Emulation, 8
- Georgia Tech Network Simulator, *see* GT-NetS
- Globlal Mobile Information Systems Simulation Library, *see* GloMoSim
- GloMoSim, 5
- GTNetS, 5
- GUI, 23, 24
- iNSpect, 10
- Integrated NS-2 Protocol and Environment Confirmation Tool, *see* iNSpect
- Listener, 25
- mediator, 18, 21
- NAM, 6, 10
- NetVis, 10, 28, 32
- Network Animator, *see* NAM
- Network Simulator-2, *see* NS-2
- NS-2, 5, 37
- omnet, 5
- OPNET, 5
- Optimized Network Evaluation Tools, *see* OPNET
- protocol, 22
- Scenarios, 46
- Simulated Visualization, *see* SimVis
- SimVis, 18, 24, 34, 42
- TCL, 6, 8, 36, 43
- Tool Command Language, *see* TCL
- Transmission Control Protocol/Internet Protocol, *see* TCP/IP
- VisSync scheduler, 30, 46



# REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

<b>1. REPORT DATE</b> (DD-MM-YYYY) 21-03-2009		<b>2. REPORT TYPE</b> Master's Thesis		<b>3. DATES COVERED</b> (From — To) Sept 2007 — Mar 2009	
<b>4. TITLE AND SUBTITLE</b>  Enhancing the NS-2 Network Simulator For Near Real-Time Control Feedback and Distributed Simulation Breaks				<b>5a. CONTRACT NUMBER</b>	
				<b>5b. GRANT NUMBER</b>	
				<b>5c. PROGRAM ELEMENT NUMBER</b>	
<b>6. AUTHOR(S)</b>  Weir, John S., Maj, USAF				<b>5d. PROJECT NUMBER</b> JON # ENG-09-200	
				<b>5e. TASK NUMBER</b>	
				<b>5f. WORK UNIT NUMBER</b>	
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b> Air Force Institute of Technology Graduate School of Engineering and Management 2950 Hobson Way WPAFB OH 45433-7765				<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>  AFIT/GE/ENG/09-47	
<b>9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b>  Dr. Robert Baonneau (703) 696-9545 robert.bonneau@afosr.af.mil Air Force Office of Scientific Research 875 N. Randolph, Ste. 325, Rm. 3112 Arlington, Virginia, 22203				<b>10. SPONSOR/MONITOR'S ACRONYM(S)</b> AFOSR/NM	
				<b>11. SPONSOR/MONITOR'S REPORT NUMBER(S)</b>	
<b>12. DISTRIBUTION / AVAILABILITY STATEMENT</b>  Approval for public release; distribution is unlimited.					
<b>13. SUPPLEMENTARY NOTES</b>					
<b>14. ABSTRACT</b>  A network simulator coupled with a visualization package enables the human visual system to analyze the results of network modeling as a supplement to analytical data analysis. This research takes the next step in network simulator and visualization suite interaction. A mediator (or run-time infrastructure (RTI) in the literature) provides researchers the potential to interact with a simulation as it executes. Utilizing TCP/IP sockets, the mediator has the capability to connect multiple visualization packages to a single simulation. This new tool allows researchers to change simulation parameters on the fly without restarting the network simulation.					
<b>15. SUBJECT TERMS</b>  Network, Visualization, Network Simulator, Simulation Feedback, Distributed Simulation					
<b>16. SECURITY CLASSIFICATION OF:</b>			<b>17. LIMITATION OF ABSTRACT</b>  UU	<b>18. NUMBER OF PAGES</b>  101	<b>19a. NAME OF RESPONSIBLE PERSON</b> Lt Col Stuart Kurkowski, PhD, ENG
<b>a. REPORT</b>  U	<b>b. ABSTRACT</b>  U	<b>c. THIS PAGE</b>  U			<b>19b. TELEPHONE NUMBER</b> (include area code) (937) 785-3636, ext 7228, stuart.kurkowski@afit.edu